An agent-based visualisation system.

Roard, Nicolas

AN AGENT-BASED VISUALISATION SYSTEM

by

Nicolas Roard

A thesis submitted to the University of Wales in candidature for the degree of Philosophiæ Doctor

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

December 2007

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed ..... (candidate)

Date 12/11/2008

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed .. (candidate)

Date 12/11/2008

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed .. (candidate)

Date 12/11/2008

Summary

This thesis explores the concepts of visual supercomputing, where complex distributed systems are used toward interactive visualisation of large datasets.

Such complex systems inherently trigger management and optimisation problems; in recent years the concepts of autonomic computing have arisen to address those issues.

Distributed visualisation systems are a very challenging area to apply autonomic computing ideas as such systems are both latency and compute sensitive, while most autonomic computing implementations usually concentrate on one or the other but not both concurrently.

A major contribution of this thesis is to provide a case study demonstrating the application of autonomic computing concepts to a computation intensive, real-time distributed visualisation system.

The first part of the thesis proposes the realisation of a layered multi-agent system to enable autonomic visualisation. The implementation of a generic multi-agent system providing reflective features is described. This architecture is then used to create a flexible distributed graphic pipeline, oriented toward real-time visualisation of volume datasets. Performance evaluation of the pipeline is presented.

The second part of the thesis explores the reflective nature of the system and presents high level architectures based on software agents, or visualisation strategies, that take advantage of the flexibility of the system to provide generic features. Autonomic capabilities are presented, with fault recovery and automatic resource configuration.

Performance evaluation, simulation and prediction of the system are presented, exploring different use cases and optimisation scenarios. A performance exploration tool, Delphe, is described, which uses real-time data of the system to let users explore its performance.

Acknowledgements

As I recall it, it all started so simply. Four years ago, I completed my master at Paris 6 University, and was keen on pursuing a Ph.D. in the same field — distributed computing. I opened a web browser, typed google.com in the address bar, and found an interesting research project proposing to combine distributed computing with graphic systems, on the Swansea University computer science web site.

I promptly sent a mail enquiring about it. Needless to say, I then embarked on a journey that reserved a lot of surprises! I moved to Swansea in January 2004, and slowly thought of it as home. Now, four years later, I find myself working at the same company that indirectly helped started all of that, acclimating myself to a new city, London.

During those four years at Swansea University, I met many people from all parts of the world. I want to thank my fellow postgraduate students: Alfie Abdul-Rahman, Will Harwood, Simon Walton, Joanna Gooch, Ben Spencer, Tim Lewis, Lidia Oshlyansky, Lyndsey Clarke, for their friendship and help. Will helped organize many fantastic evening for us, as well as being one of the creator of the famous labminton !

I want in particular to thank David Chisnall — we had so many fruitful discussions on varied topics ranging from user interface to language theory, he helped me rethinking and refining a lot of my ideas about computer science.

A lot of other people made Swansea home — Dev and Joy, Teiko, Alberto, Roberto, Tom, Rachel, Andy, Bash, Matt, Ben, Dave, Markus, Jérémie...

My friend Gorka Lasso Cabrera shared more than its lot of coffees, pints and discussions with me — again, I don't think I would have gone through all this without your constant friendship and support. You did a lot to make Swansea a fantastic place !

Many other friends helped by being there all those years, through episodic visits in France and the wonders of internet. I want to thank Brice, Marie, Max, Mél, Kim, Piu, Emeric, Jo, Christian, Doud, Jean, Quentin, Damien...

I want to thank my supervisor, Dr Mark Jones, for his patience, support, and understanding during those years — he helped me being a much better researcher, and never balked at my crazy ideas but helped me defined them. I also want to thank Pr Min Chen — his support meant a lot this last year.

I also want to thank my manager at Google, Dr Dave Burke, for his support during those last few months, as well as my colleagues Andrei Popescu, Ben Murdoch, Tim Cox, Steve Block and John Ripley. Finishing to write a Ph.D. thesis while working in parallel is definitely not an easy mix, and they coped with me during it...

Many thanks to my friends David Chisnall, Damien Pollet, Quentin Mathé and Alfie Abdul-Rahman for reading some preliminary drafts of this thesis and giving me insightful suggestions and comments.

A Ph.D. is both a solitary journey and something that cannot be done without the help, love and care of so many people. To all of you, thank you.

Finally, without the support and love of my girlfriend, completing this work would have been a much more difficult undertaking. Emanuela, you were always here, through all that. I love you. You make me a better person.

I want to finish by dedicating this thesis to my family — my parents, Francis and Laura, my brother Sébastien, my sister Amandine, my grandparents, Clément and Dédé, and of course all my cousins, aunts and uncles ... You were with me and in my thoughts every day all these months, I would not have been able to finish without you. This thesis is yours.

Contents

List of Figures

List of Tables

Listings

Pour Francis, Laura, Sébastien et Amandine

Introduction

(CHAPTER ...1)

Introduction

Contents

> *Alexander Adell and Bertram Lupov were two of the faithful attendants of Multivac. As well as any human beings could, they knew what lay behind the cold, clicking, flashing face – miles and miles of face – of that giant computer. They had at least a vague notion of the general plan of relays and circuits that had long since grown past the point where any single human could possibly have a firm grasp of the whole. Multivac was self-adjusting and self-correcting. It had to be, for nothing human could adjust and correct it quickly enough or even adequately enough – so Adell and Lupov attended the monstrous giant only lightly and superficially, yet as well as any men could.*
>
> — Isaac Asimov, *The Last Question*, 1956

FOR A LONG TIME, computers were thought of as large, massive, and few. Isaac Asimov, with his fictional supercomputer *Multivac*[1], convey an idea of the computer that was the reality of the time. Indeed, as late as 1977, Ken Olsen, a pioneer of the computer industry[2], famously said that "There is no reason for any individual to have a computer in his home"[3].

The micro-electronics revolution, leading to the introduction and widespread ac-

[1] A play on UNIVAC, the first commercial computer made in the United States (1951)
[2] Founder and CEO of Digital Equipment Corporation
[3] In a meeting of the World Future Society in Boston (1977)

ceptance of the micro-computer in the 1980s turned around that idea. Instead of this vision of few, omnipotent, centralized mainframes as illustrated by Asimov, we now have large numbers of increasingly smaller and more accessible computers, distributed along local and global networks, communicating to work together.

Yet, while Asimov's vision of a single mainframe did not happen[4], he nonetheless highlighted in this short story a very tangible problem we encounteer every day: the sheer complexity of managing all those computers to make them work together on common tasks, and the necessity for complex computer systems to be autonomic, self-adjusting, self-correcting.

In an ironic example of reality rejoining fiction, IBM is one of the first companies to have recognised the problems of making a very large number of computers and software systems cooperate and integrate, and to have proposed a common vision toward solving this: Autonomic Computing (AC).

In a late 2001 manifesto [126], IBM observed that the main obstacle to progress in the IT industry was a looming software complexity crisis, with applications and environments weighing tens of millions of lines of code, requiring weeks by teams of skilled IT professionals to be configured and tuned. IBM's proposal, by its senior vice president of research, Paul Horn, during a March 2001 keynote address at the National Academy of Engineers at Harvard University, was to build software systems modelled on the human body, as autonomic components.

The principal characteristics that such an autonomic system should display would be, according to IBM, the following:

- self-configuration: the system should conform to high-level policies, the detail of it should be dealt with automatically
- self-optimisation : the system should try to find opportunities to improve its performance
- self-healing: the system should detect and repair problems
- self-protection: the system should defend itself automatically against attacks or cascading failures

[4] Although one could argue that *Multivac* is just another name for what we now call the Internet!

1.1 Objectives

The overall research objective corresponds to the following question: *How can we build an autonomic visualisation system ?*

Visualisation on its own is a complex process, touching a wide range of problems — rendering algorithms or architectures, human-computer interaction or collaboration.

High-performance visualisation, a derivative of High-performance computing (HPC), wants to apply HPC techniques to the visualisation field. Indeed, visualisation can be a very intensive process, with researchers and users working with very large datasets, rendered using complex algorithms. While workstations' computational and graphical capabilities improve continuously, so do the datasets' size and visualisation process.

Using HPC techniques — with cluster of machines working together in a distributed system to implement a graphic pipeline — allows to run more complex rendering algorithms and larger datasets.

But as with any complex software system, it should not come as a surprise that we encounter the same type of problem IBM highlighted in their autonomic computing manifesto.

In fact, it could be argued that a visualisation system is one extreme case of autonomic computing, as in addition to these shared problems, it also begs for low latency in order to provide an interactive visualisation system for its users — an added constraint on already complex demands.

The perspective developed in this thesis thus tries to answer the following question: *how can we create a software system that tends toward autonomy, while maintaining the flexibility and high performance needed by the visualisation process ?*.

Our proposed solution to this question is to develop a multi-agent system to implement a distributed graphic pipeline. The system needs to be open and extensible, to allow a high degree of flexibility, necessary to implement complex visualisation processes and the autonomic features we need.

The general approach is thus to build a reflective system, that is, a system that can be modified by itself, which in our experience allows the most flexibility. An expected problem with developing a reflective, open system, is to reach good performance while keeping this flexibility. Considering that visualisation needs to achieve interactive frame rate — above five image seconds (*i.e.* a complete iteration needs to be done in under 200 ms), with twenty-five images second a worthy goal, (*i.e.* iterations under 40 ms) — to be usable, this is a non-trivial problem, particularly as we are working in a distributed environment which by nature introduce delays. An important aspect of our performance goals is also to have a latency as minimal as possible; even a very flexible, sixty images per second system that would react

to changes with a five minutes delay would be near completely useless in terms of real-time interaction with users.

The different aspects of an autonomic system as defined by IBM — self-configuration, self-optimisation, self-healing and self-protection — should be implemented as agents, allowing for greater flexibility.

In order to provide self-optimisation, agents should record past performance, and simulations of the system need to be developed.

1.2 Thesis outline

The thesis is divided into three main parts; the first part consists of Chapters 2 and 3, which will present a literature review in distributed visualisation, system management, fault tolerance and performance evaluation.

The second part, composed of Chapters 4 and 5, will respectively present the design of a generic multi-agent platform and the implementation of a distributed graphic pipeline using this platform.

The third part of the thesis will describe a reflective, autonomic system, built on top of the agent platform (Chapters 6 to 8).

In Chapter 2, we present a broad overview of parallel systems, followed by a specific overview on the technologies of visualisation, and an introduction to visual supercomputing ideas.

Chapter 3 discusses the problems of managing complex systems such as large distributed systems. Distributed systems concepts will be presented, as well as types of failures and approaches for fault tolerance. Different metrics used for the evaluation of parallel systems are described. Finally, the grid initiative and the autonomic paradigm are presented, and the e-Viz project outlined.

Chapter 4 concerns the design of an agent system, and will present our general approach to designing an autonomous system based on a multi-agent system, aside of the specific problems of visualisation systems. The different agent mechanisms provided by the system, such as an implementation of Map/Reduce, will be presented.

Chapter 5 will describe how a distributed graphic pipeline can be implemented on top of the agents system described in the previous chapter, and talk about some of the implementation details. A performance evaluation of the pipeline is presented.

Chapter 6 will make the case for a reflective pipeline, and will then discuss how those reflective features were implemented. Examples of reflective patterns will be introduced, as well as how autonomic features can be implemented with the system.

Chapter 7 will describe additional reflective patterns, applied to the specific problems of visualisation, and notably will describe how Map/Reduce can be adapted to real-time rendering.

Chapter 8 will introduce different simulation agents used to provide performance estimations at run-time, and how simulation models can be useful to identify bottlenecks as well as configuring the system to be as efficient as possible.

Finally, Chapter 9 will provide a conclusion, highlighting the contributions of this thesis and introducing some possible future research subjects.

In addition to the Chapters forming the body of the thesis, Appendix A will present Delphe, a simulation application we created to explore run-time and post-mortem performance data. Appendix B will describe how we implemented scripting support using the Smalltalk language, allowing for quick prototyping of agents. Finally, Appendix C will present a steering widget we implemented in different environments.

PART I

Literature review

(CHAPTER ...2)

Visual Supercomputing

"A display connected to a digital computer gives us a chance to gain familiarity with concepts not realizable in the physical world. It is a looking glass into a mathematical wonderland."

— Ivan E. Sutherland

Contents

In 1965, Gordon Moore observed in a famous article [182] that the number of elements on an integrated circuit doubled roughly every 24 months (for a given investment). This became known as Moore's Law, and holds true even today [1] (Figure 2.1 on the next page shows a graph of the predicted and actual transistor count in Intel processors since 1971). While performances certainly depend on more than the number of transistors used (*e.g.*, different CPU architectures can be more efficient than others given the same number of transistors), it nonetheless offers a metric of the performance increase in processing power over the years [2].

Not only have performances increased tremendously, costs have followed an opposite trend, with computers becoming affordable by anyone. As a comparison, the Cray-1, one of the first available supercomputers, launched in 1976, cost more than $8 million, and had a theoretical performance of 160 million of instructions per

[1] It is debatable that this "Law" became a self-fulfilling prophecy for the industry.

[2] Processing power measured in MIPS does indeed follow the same exponential trend.

second (MIPS). The Intel Core 2 processor launched 30 years later, cost less than $200, with a theoretical performance of 57063 MIPS.

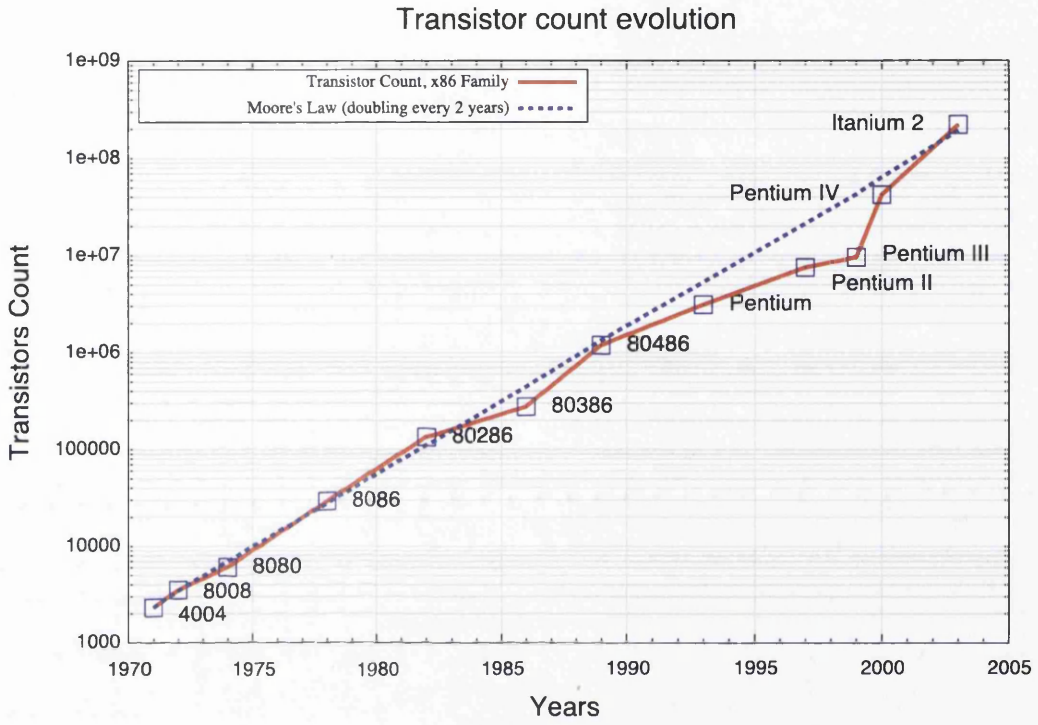


Figure 2.1: *Moore's Law : transistor count doubling every 24 months*

Yet, while this historical increase in CPUs' performances is particularly impressive, it is lagging behind the requirements for the most computationally expensive tasks of the day; as in a modern-day reenactment of Sisyphus' myth, those tasks are continuously redefined, with more computationnal power only unlocking new areas where computers can be used. In short, expectations are increasing faster than available processing capabilities.

Computer users are now used to do real-time processing of large volumes of data (complex simulations, 3D games, home cinema, internet), while a mere 15 years ago expectations were much lower (introduction of the first CD-ROM games, small video resolutions). Scientists face the same problems, albeit on a bigger scale, with the increase of raw computational power opening new research areas. More and more data is gathered or generated, and needs to be data-mined and visualised, possibly in real-time. The question we can ask ourself is how is it possible to achieve this kind of processing, if the fastest CPU available is not fast enough?

If we look at computers' performance, an obvious limitation is their architecture. Historically, computers are centralized systems — one central processing unit (CPU) associated to memory banks, communicating with various peripherals. By definition, this architecture led to processing capabilities directly limited to the performance of

the single CPU used by the computer.

In the mid-eighties, the combination of two innovations led to a departure from the traditional model of centralized systems: first, the introduction of fast local networks (using technologies such as Ethernet [176]) and second, the availability of fast, (comparatively) cheap machines based on powerful microprocessors (Moore's Law at work). These two innovations jointly changed the way computers were used, as instead of sharing a single expensive central computer (mainframe computing), every user could have its own workstation, paving the way to the personal computer revolution.

The improvements in microelectronics allowed new designs: most notably, the Connection Machine [122, 123] was a single machine with up to 65536 (weak) CPUs made available in the late 1980s, and showed the attractiveness of parallelisation to improve performances — as well as its difficulties, particularly in adapting algorithms to this architecture.

A stronger impact was felt from the introduction of fast local networks connecting workstations, which led to the widespread adoption of *distributed computing*: connecting computers to combine their power, creating clusters of computers able to perform tasks ordinarly reserved to supercomputers, at a cheaper cost (an approach typically referred as high-performance computing or HPC). The Internet is a direct descendant of this networking revolution. Recent supercomputers' installations follow the same approach, by clustering machines interconnected using specialised communication backplanes or very fast network links. BlueGene/L is for example the world's largest system, with 131072 processors and a peak rate of 367 Tflop/s [12, 237]. We can say that distributed computing permit a better use of existing computational resources, and vastly outperforms the performances of a single CPU, becoming a way to progress further than Moore's law.

It is notable that while Moore's Law seems to still be valid for now (current estimates for it to end range from 10 to 20 years using our current technology prospects), the actual *speed* increase of individual CPUs is starting to stagnate. Reducing the transistors' size allows more transistors and higher frequencies; but this in turn significantly increases the heat generated, thereby limiting the practical frequency used.

The solution brought by the industry is the creation of multi-core CPUs as, while frequency is harder to increase, the size reduction of transistors permits more cores per CPU for the same die area.

Dual-core CPUs are now extremely common and 8-cores easily available. Intel for instance aims to deliver 32 cores CPU by 2010 [3], and has already demonstrated research prototypes processor with 80 cores on a single die [4].

[3] `http://www.xbitlabs.com/news/cpu/display/20060710072810.html`

[4] "To demonstrate how Moore's Law will continue well into the future with amazing potential, Otellini showed a new research prototype processor that has 80 floating point cores on a

No matter what, the future will lead us to more parallelised systems rather than ever more powerful single CPU.

Yet, though distributed computing is becoming more and more prevalent, it is far from being a solved problem. Distributed computing has its own set of problems, particularly the difficulties of distributing work, synchronisation mechanisms, communication speed and latency — all taking their toll, making performances growing sub-linearly with the number of computers used, a fact illustrated by Amdahl's law [16].

The following sections will discuss in more detail Distributed Computing, the Visualisation process, and how High-Performance Computing is in fact needed for Visualisation.

2.1 The Need for Visual Supercomputing

Visualisation is a common task of today's computers and devices. Starting in the 1980s, personal computers offered graphical user interfaces (GUIs) using bitmapped displays. Accelerated graphics hardware started to appear, first for 2D, then 3D work, on specialised graphics workstations at the beginning of the 1990s. But the democratisation of personal computers triggered the democratisation of graphics hardware, often lead by video games: advanced graphic technologies once reserved to the high-end workstation market moved to desktop computers.

Today, the result of this process is that a large variety of visualisation resources are available. Modern desktop computers allow complex visualisation tasks. While many users are satisfied with their visualisation capabilities, some users are more demanding; complex visualisation rendering processes still cannot be done in real-time easily, and very large data sets still rely on high performance computing facilities to be visualised.

In addition to these computational limits, the way visualisation systems are now employed is also shifting; mobile computing systems are more and more prevalent (mobile phones, Internet tablets, tablet PC, laptops, personal digital assistants), and such systems generally do not benefit from powerful graphics accelerators, and have to rely on external resources in order to fulfill complex visualisation tasks.

Those different needs — the increase in visualisation dataset size (data mining), the need for complex visualisation algorithms (volumetric datasets) or for instantaneous availability of large datasets (virtual worlds), and new mobile use cases — cannot be fulfilled with the existing visualisation infrastructure, largely based on desktop computers, which can hardly scale up.

single die.", Intel press release, 26/09/2006, `http://www.intel.com/pressroom/archive/releases/20060926corp.htm`

This leads to a series of questions introduced in [35]:

- *What would be an adequate infrastructure?*
- *In what way do the computational requirements of visualisation differ from other software technologies?*
- *Is it desirable or feasible to bring a range of technologies under one management (not necessarily under one roof)?*
- *If it were feasible to build such an infrastructure, what would be an appropriate virtual machine interface for the infrastructure?*
- *How should users' experience be managed when they access visualisation resources in the infrastructure?*

The computer graphics and visualisation community has in fact been seeking answers for these questions for the past few decades. Notably, a huge amount of effort has been spent in developing specialised graphics hardware and in taking advantage of the latest technologies for high performance computing. A large body of research exists on parallel, distributed and mobile visualisation techniques. However, in general, these questions were addressed with a fragmented perspective, not using an holistic approach that would consider a complete system.

2.2 Parallel Systems

Visual supercomputing can be seen as a subset of distributed computing and parallel systems, borrowing a lot of techniques and concepts from those fields. It is therefore necessary to introduce some important notions of parallel computing before discussing in more details visual supercomputing.

We will first describe the different paradigms of parallel computation, and introduce common taxonomies used to describe parallel systems. We will then present parallel programming paradigms, computer architectures and models of inter-process communications.

2.2.1 Taxonomy of Parallel Systems

System taxonomies allow us to classify different types of systems and architecture in a common hierarchy. In addition to permitting meaningful comparison of different systems, it gives us a common set of concepts and vocabulary. We present in the following sections the most common taxonomies used to describe parallel systems.

2.2.1.1 Flynn's Taxonomy

In 1972, Flynn's taxonomy [87, 86] redefined parallel architectures, and whilst it may be a little outdated now, it is still generally appropriate and widely used.

	Single Instruction	Multiple Instructions
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Table 2.1: *Flynn's taxonomy*

Flynn identifies four categories (table 2.1):

1. SISD, or Single Instruction, Single Data stream
2. MISD, or Multiple Instruction, Single Data stream
3. SIMD, or Single Instruction, Multiple Data steams
4. MIMD, or Multiple Instruction, Multiple Data streams

SISD simply characterises a sequential computer without any kind of parallelism, such as old personal computers (modern CPUs do include various parallelisation stages and SIMD units, and modern operating systems let programs be run in parallel).

MISD is a much more unusual architecture – if we have multiple instructions we usually work with different kind of datastreams (i.e., MIMD). Some redundant systems (avionics) could conceptually be classified as MISD, although only one "true" MISD computer existed (the Carnegie-Mellon University's C.mmp multiprocessor).

For distributed computing, the principal categories identified by Flynn are SIMD and MIMD; interestingly, Flynn introduced those categories before such systems existed. The introduction of Flynn's taxonomy helped clarify what a "parallel system" was.

SIMD: Single Instruction, Multiple Data

SIMD means that we perform the same single instruction on multiple data. A canonical example would be some vector multiplication (each parallel processor could handle a different component of the vector). A theoretical model would be of a control unit that would send at every cycle a common instruction to each processor to execute.

While SIMD machines started with supercomputers like the Cray XMP, today, the SIMD architecture is mostly found in small, embedded processing units like graphi-

cal processor units (GPUs), digital signal processors (DSPs), or the various "vector" units introduced by CPU founders (*e.g.*, Intel's SSE, Motorola's AltiVec).

MIMD: Multiple Instructions, Multiple Data

MIMD means that multiple processors execute different instructions on different data, at the same time. The processors work independently and asynchronously – distributed computing as mentioned in this chapter's introduction is generally classified as a type of MIMD.

The set of processors involved in MIMD systems are usually identical (symmetric multiprocessing (SMP) settings), more rarely different (*e.g.* with heterogeneous clusters, asymmetric multiprocessing (AMP) settings).

MIMD machines work with shared memory (each individual processor accessing the same memory) or distributed memory (each individual processor having its own memory). In clusters, a virtual shared memory system is often provided, where each node accesses the memory as if it was shared, while it is in reality implemented using the different individual memory units attached to each processing unit.

2.2.1.2 Extensions of Flynn's Taxonomy

MIMD architectures can be further divided in the following categories:

1. SPMD, or Single Program, Multiple Data streams
2. MPMD, or Multiple Program, Multiple Data streams

SPMD [68] is conceptually close to SIMD, the main difference being that programs are used rather than instructions. In the SPMD model, a fixed number of identical programs are executed concurrently, working on a different part of a dataset.

MPMD architecture [51, 52] is a more generic model than SPMD, as different programs can be created dynamically and run concurrently to process the data.

2.2.2 Models of Parallel Computation

Computation models are abstract models that allow programmers and researchers to compare different platforms and algorithms. Parallel computation models are abstract models of parallel architectures; historically a difficult task as parallel computers widely varied in their architecture. In this section, we present some of the common models used in the literature. More detailed information about parallel computation models can be found for example in the thorough review by Skillicorn and Talia [225].

2.2.2.1 PRAM Model

The parallel random access machine (PRAM) model [88] is one of the early model of parallel machines. This basic model proposes a set of independant processors, using a shared global memory, and synchronised on the same clock. Processors have access to the entire memory, but only one processor can access a specific location during the same step. A rather low-level specification of the program steps is required to use the PRAM model.

This model is notable for having been used for a lot of early theorietical analysis of parallel computation. Nonetheless, the over-simplification of this model, notably in the way it models memory accesses cost and latency, makes it inadequate for accurate algorithms cost prediction.

2.2.2.2 BSP Model

The bulk synchronous parallelism (BSP) [251] is a higher-level model presented by Valiant in 1990, trying to capture the characteristics of a parallel architecture using a few parameters.

A BSP machine consists of a set of P processors with local memory. Those processors are interconnected through a network, modelled via two parameters, its latency l when doing doing a synchronisation step (barrier synchronisation) and its data rate g. The BSP moddel assumes hardware support for the synchronisation step. Those parameters are usually determined experimentally.

A BSP program uses p threads, and is organized in *supersteps*, where a superstep consists in the following actions:

1. computation on each processor using local values
2. global message transmission from each processor to any other groups of processors
3. barrier synchronisation

The cost of the program is easier to compute with the BSP model, as we can use the maximum computation time during a step ω and the maximum numbers of values sent or received by any processors θ to get the total time for a superstep t such as:

$$t = \omega + \theta g + l$$

2.2.2.3 LogP Model

Culler *et al.* [62] introduced the LogP model in 1993. This model shares much of its goals with BSP, attempting to capture the bottlenecks of parallel machines using a

few parameters:

- L : the communication delay (latency)
- o : the communication overhead
- g : the communication bandwidth
- P : the number of processors

A notable difference with BSP is that LogP does not assumes special hardware support for synchronisation — rather, all synchronisations steps are done through messages. Another difference with BSP is that in the LogP model, processors work asynchronously, so that a processor can use a message as soon as it arrives (in BSP, a processor would need to wait for the next superstep).

LogP tries to strike a balance between details, accuracy and simplicity. One of its advantages is that it avoids specifying the programming style or the communication protocol, and is thus equally applicable to various parallel programming paradigms.

2.2.3 Parametric Models

The theoretical models we introduced so far are based on the systems' characteristics they are modelling; even with the high-level approach taken by BSP and LogP, a certain amount of knowledge about the modelled systems and their performances is necessary.

An important use case of having a model of a system is to use the model to evaluate the system's performances, or analyse how a specific algorithm or software architecture will behave on a given sytem; more specifically, the model can be used to predict performances.

A rather different approach to those detailed models is to use parametric models [16, 62, 109, 110] that only provide a generic performance law, irrespective of the specific architecture of the observed system. Such parametric models are black-box models, which can be used to experimentally measure and predict a system's performances; this experimental nature make them ideal candidate to observe the behaviour of complex systems such as the one used in Visual Supercomputing, using only a very limited observable knowledge of the system.

We introduce in the following sections two parametric models that we will use to observe performances in our system.

2.2.3.1 Amdahl's Law

Amdahl's law [16] is a simple formula that is used to model the performance of a system, when just parts of the system are improved. Specifically it is often used for

modelling distributed systems. The general formula gives S, the speedup factor, such as:

$$S = \frac{1}{\sum_k^n \left(\frac{P_k}{S_k}\right)} \tag{2.1}$$

where P_k is a percentage of instructions that can be improved (or slowed), S_k is the speedup factor for these instructions, k is a label for each different percentage and speedup, and n is the the number of total speedup/slow-downs comprising the total time.

Adaptation to Parallel Computing

For parallel computing, we can rewrite the equation 2.6 by using two couples of (P_k, S_k) — one $(k = 0)$ representing the linear, non-parallelisable part of the system (where $S_0 = 1$) and the other $(k = 1)$ representing the parallelisable portion, where S_1 varies depending on the number of nodes. Therefore:

$$S = \frac{1}{\frac{P_0}{S_0} + \frac{P_1}{S_1}} = \frac{1}{P_0 + \frac{P_1}{S_1}} \tag{2.2}$$

$$\tag{2.3}$$

As P_1 represent the parallel part of the system, and P_0 the non-parallel part, we could rewrite S as:

$$P_0 = 1 - P_1 \tag{2.4}$$

$$S = \frac{1}{(1 - P_1) + \frac{P_1}{S_1}} \tag{2.5}$$

Expressing the equation in terms of number of nodes N and with P the parallel part of the system, we derive the classic form of Amdahl's law applied to parallel computing, where $S(N)$ is the speedup factor for N nodes:

$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}} \tag{2.6}$$

The speedup factor is a widespread metric used to describe and evaluate parallel systems [113, 222]. While important as a guide, we should note that Amdahl's law only describes an ideal system, which does not account for many additional overheads in a distributed system.

2.2.3.2 Universal Model

Amdahl's law, while a useful tool, is an optimistic model of a parallel computation: there is no possibility to get worse results with more nodes — at worst performance will stay flat. In real systems, it is not uncommon to get *worse* performances by adding more nodes; the amount of time spent in synchronisation and management tasks outwheights the speedup gained by more parallelisation. An interesting parametric model is the universal model described in [111, 110, 109], which contains Amdahl's law as a special case, and model a potential worse performance when adding nodes.

The model's equation is the following:

$$S(N) = \frac{N}{1 + \alpha(N-1) + \beta N(N-1)} \tag{2.7}$$

Where $S(N)$ is the speedup factor, $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$. The term α is the percentage of time not parallelised, *i.e.* $\alpha = 1 - P$. For $\beta = 0$, the equation is equal to the speedup equation from Amdahl's law:

$$\begin{aligned} S(N) &= \frac{N}{1 + \alpha(N-1)} \\ S(N) &= \frac{N}{1 + (1-P)(N-1)} \\ S(N) &= \frac{1}{(1-P) + \frac{P}{N}} \end{aligned}$$

2.2.4 Hardware Architectures

A non-parallel computer (figure 2.2 on the following page) architecture can be simplified into:

- a CPU unit
- a local memory system (RAM)
- a local disk system

With the CPU unit itself being composed of the following subsystems:

- the actual CPU
- CPU cache unit (often on the same die as the CPU) [5]
- the memory management unit (MMU)

[5] CPU caches are small memory units (compared to the machine's RAM), but with a much faster access and/or bandwidth (usually due to the proximity with the CPU)

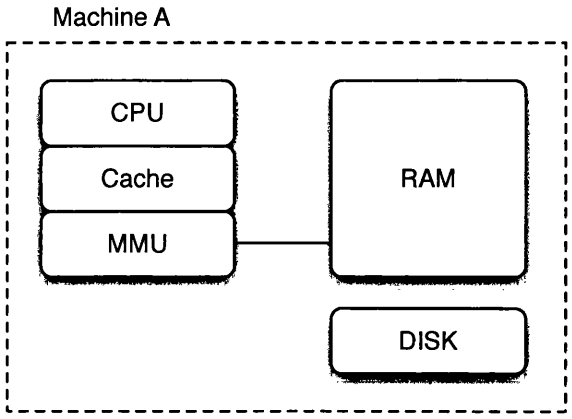


Figure 2.2: *Standalone system*

Parallel machines retain the same overall components — but of course are composed of multiple instances of those components. We can classify parallel machines hardware depending on which resources are shared, and how they are shared.

2.2.4.1 SIMD Processors

While modern parallel machines are architectured around multiple processors (often based on clusters), one of the first widely used instance of parallelisation in hardware came with vector processors in supercomputers such as the Cray-I [197].

Vector processors execute multiple CPU instructions in parallel on data organized in vectors; scalar processors on the other hand execute one instruction at a time.

A typical case of vector processing is code operating on loops, where each loop is independant of the other; each loop can then be executed in parallel rather than executed one after the other. Specialised compiler were able to automatically vectorise code [148].

Modern processors are a combination of scalar processors and vector processors, with specialised vector units embedded, such as Intel's MMX or IBM and Motorola's AltiVec.

2.2.4.2 Multiple Processors Architectures

A multiprocessor setup consists in having in the same machine multiple CPUs (figure 2.3 on the next page), with the CPUs communicating via a fast local bus. This setup allows fast communication and synchronisation between the CPUs. This

type of architecture corresponds to MIMD, with multiple processors functioning asynchronously and independently.

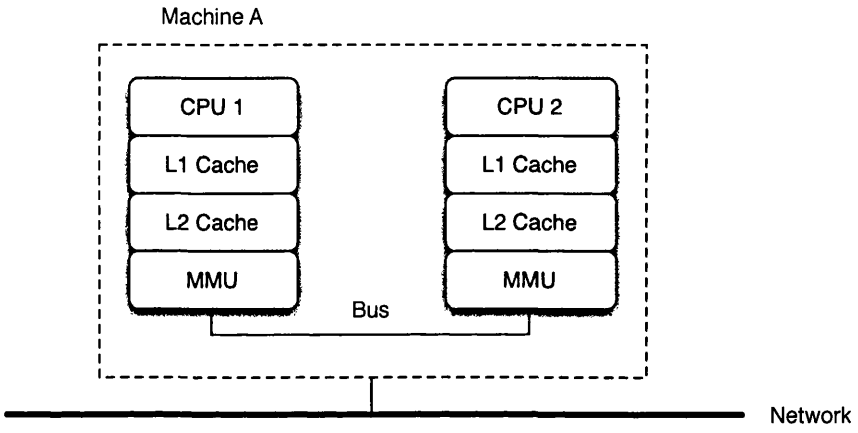


Figure 2.3: *Multiprocessor system*

A recent twist on the multiprocessor setup is the multicore approach (figure 2.4). Instead of having fully complete CPUs, the CPUs can share parts of their architecture, such as their level 2 cache and their memory management unit. Compared to complete CPUs, this is a more efficient use of die area, for similar performances. Multicore setup are usually exposed to operating systems as separate CPUs, so normal multiprocesses programming techniques can be used.

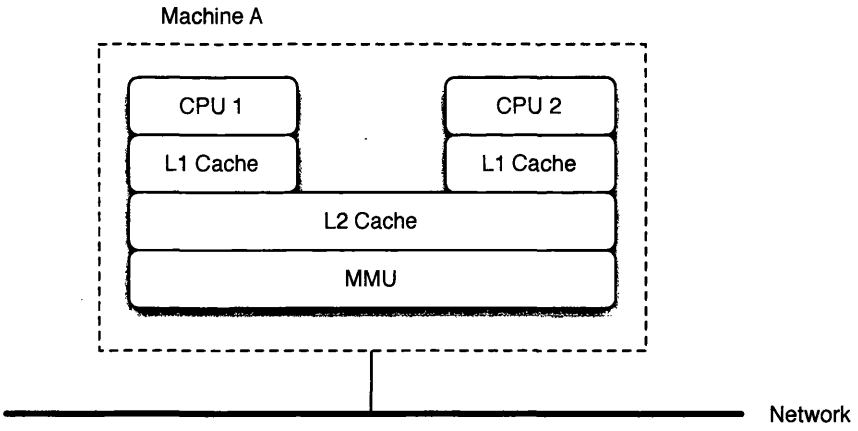


Figure 2.4: *Multicore system*

Finally, the last common architecture consists in having full separate computers arranged in a cluster (figure 2.5 on the next page). Most recent supercomputers (*e.g.* [237]) are organized in that manner. While less efficient than the previous methods — as the communications between CPUs have to pass by the (comparatively) much

slower network relying the individual machines — this method has the benefit of allowing easy scaling of the system, without having to drastically change the base architecture of CPUs. In addition, nothing prevents the individual nodes to have multiprocessor or multicore setups, and following the success of multicore chips, this is now a common occurence in clusters. This configuration is of particular interest to us as we will target our visualisation system for a cluster of identical machines connected via a fast local network.

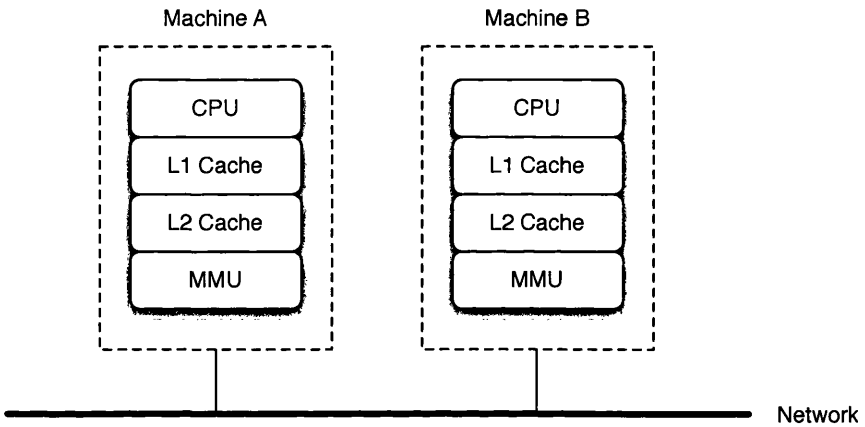


Figure 2.5: *Cluster system*

2.2.4.3 Memory Architectures

One of the strongest impact on a parallel machine – guiding its performance and the way it is programmed – is caused by the way processors are connected and communicate with the available memory.

A typical classification distinguishes non-uniform memory access (NUMA) and uniform memory access (UMA) [224]. In addition, with both UMA and NUMA systems, memory can be organised either in a distributed way (each processor having a private, fast local memory unit) or in a shared way (every processor accessing the same memory). Those two memory structures need synchronisation mechanisms to handle access to shared data. Hardware specially designed for shared memory systems provides local cache memory, kept consistent with the global memory using cache coherency procotols [243].

Memory Access

In UMA systems, all the processors access the same memory, within the same time constraints [6]. UMA systems are better known as symmetric multi-processors (SMP) systems.

NUMA systems on the other hand have different time constraints per processor while accessing the memory. NUMA systems can thus be larger and more distributed than UMA systems, as the memory can be distributed [272].

Shared Memory Systems

In shared memory systems, the processors share the same memory access; the memory is connected to the processors using dynamic interconnection networks [240].

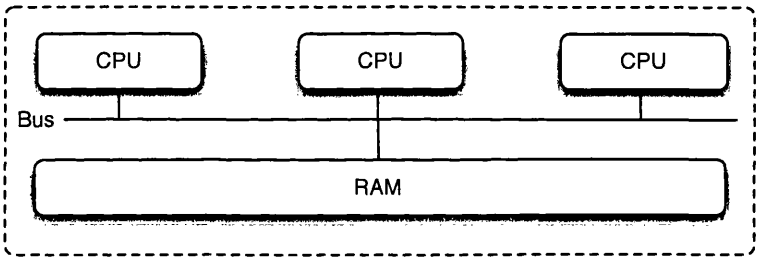


Figure 2.6: *Shared memory system*

A common architecture used to connect processors and memory units is a bus network (figure 2.6). More scalable and more complex architectures are for example a crossbar switching network (figure 2.7 on the next page), notably used in the Cray Y-MP.

A crossbar switch arbitrates access to the resources connected to it; used to access memory, it means that the CPUs do not have to manage this access (*e.g.* deal with concurrent access). Removing this management task from the CPUs accelerate the system. The main issue with such a device is its cost compared to using a bus system to connect to the memory.

Multistage interconnection networks scale better than bus networks in terms of performances, and usually scale better than crossbar networks in terms of cost. The Omega network (figure 2.8 on the following page) is an example of such multistage interconnection network, and has the property of being self-routing: the path to

[6] It is worth noting that no real non-trivial UMA systems really exist; even SMP systems may have (albeit small) different time constraints, as simultaneous write, contrary to simultaneous read, induces a delay.

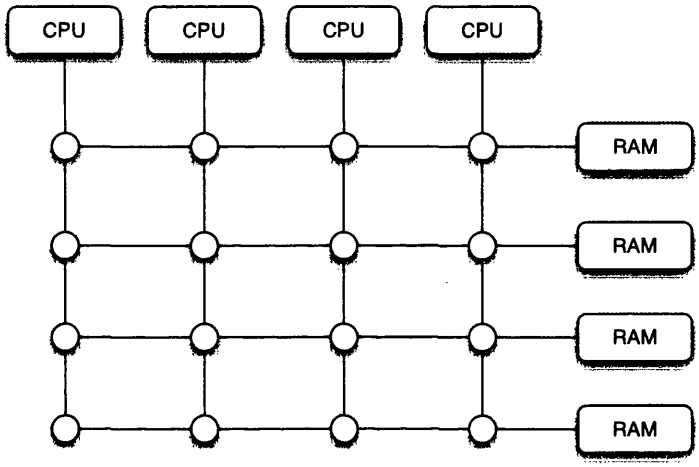


Figure 2.7: *Crossbar switch*

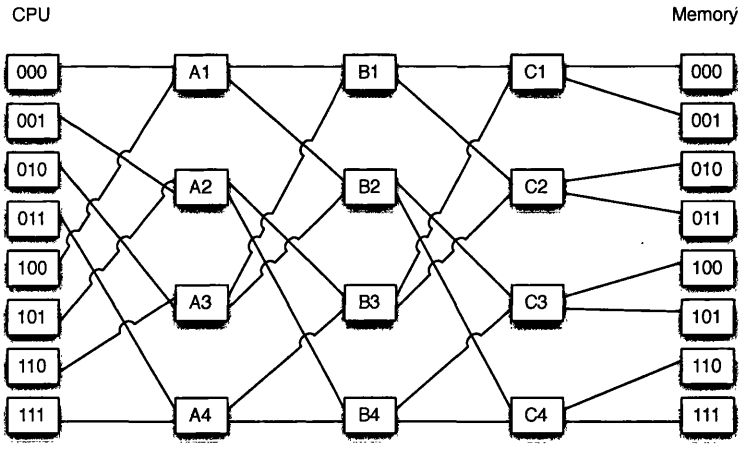


Figure 2.8: *Omega network*

reach a destination can be determined directly from the address. Stage n of the network looks at n^{th} bit of the address, and sends to the upper cell in the next stage if the bit is equal to zero, and to the lower cell if the bit is equal to one.

Distributed Memory Systems

The memory can instead be distributed on multiple machines (figure 2.9). In this configuration, each processor have a private, fast access to a local memory unit. Accessing the memory of another processor typically uses some type of message passing architecture (*e.g.* Cray T3D).

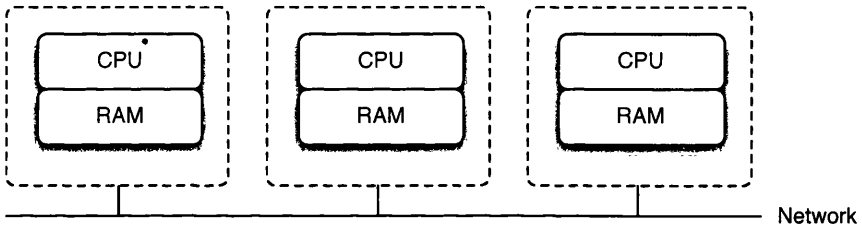


Figure 2.9: *Distributed memory system*

Distributed memory systems are often considered harder to program than shared memory systems, but they have the principal advantage of being able to scale to thousands of processors [69]. This is typically the architecture used in modern supercomputers setup.

2.2.5 Parallel Programming Paradigms

The previous section covered parallel hardware architecture. The hardware used is of course only one half of a complete architecture – the type of software, and more particularly the way the software implements parallelism, is equally important.

In fact, parallel applications can be classified according to the parallel programming paradigm they use: *data parallel* and *task parallelism*. In general, parallel applications and system are not purely data parallel or task parallel, but fall somewhere in between, borrowing concepts from both sides.

2.2.5.1 Data Parallel

The *Data Parallel* paradigm takes advantage of the parallelism of the data itself, distributing the data onto different nodes. The computation applied to the data chunks usually is identical.

Parallelising a computation using this paradigm — *i.e.* distributing the data — was historically done via a specialised compiler [148], exploiting the SIMD architectures (Vectorisation) of the existing supercomputers. Many languages were developed to support the data parallel paradigm in the late 1980s and early 1990s, such as the CM-2 family (*i.e.* C*, CM-Fortran and *Lisp by Thinking Machine corp), MP-2, Dataparallel C, DINO, PC++ and High Performance Fortran (HPF) [148].

2.2.5.2 Task Parallelism

Task Parallelism is the reverse approach, where the computation is divided into individual chunks (organised in threads or processes), then executed on different processors.

Communication between tasks is an important aspect of task parallelism. The following sections will detail possible approaches.

2.2.5.3 Shared Memory

The idea behind the shared memory paradigm is that programmers are provided with a virtual machine, acting as if the memory available was on the same machine. The physical memory can be distributed among nodes, or can be a real shared memory architecture; the programmer is isolated from the underlying hardware.

Among examples of this approach we can cite Linda [99, 44, 45], which provides language extensions to C and Fortran to support shared-memory programming. SR [19] is another language that supports both the shared-memory paradigm and the message passing paradigm. X3H5 is an ANSI standard for shared-address-space programming in the context of single program and multiple data stream (SPMD). A notable implementation is OpenMP [6, 65], supported by many commercial compilers, an API for shared-memory programming on multiprocessor architectures.

Programming environments following this paradigm usually provide facilities to create processes and threads executing in parallel, sharing memory space, as well as mutual exclusion and synchronisation mechanisms. As this paradigm corresponds to a virtual machine, it is an easier approach for programmers; but correct mutual exclusion is critical [97].

2.2.5.4 Message Passing

Message passing is a very common programming paradigm, where different processes communicate through messages. Adapted to parallel programming, it allows to have the processes running on different nodes. It requires more manual work

from the programmers — specifying subtasks to execute, start/stop them, and synchronise them. Many environments and languages allow this paradigm.

Environments and frameworks such as Message Passing Interface (MPI) [41] and Parallel Virtual Machine (PVM) provide implementations of the message passing paradigm. MPI is one of the most popular programming environments for developing parallel applications. It is hardware independent, and provides a set of library interface standards for managing process creation and message communication. The MPI-2 standard introduces dynamic process management, though only a few implementations are MPI-2 compliant at present. PVM [97] from Oak Ridge National Laboratories is another implementation of the message-passing paradigm. Using the notion of a virtual machine, PVM enables programmers to treat a set of heterogeneous computers as a single parallel computer. Although MPI is believed to be faster within a large multiple processor system, PVM still scores highly due to its fault tolerance and recovery [98]. Other vendor independent libraries for the messaging passing paradigm include EXPRESS, P4 and PICL.

In addition to environments such as MPI and PVM, some languages provide message passing as a built-in feature. Smalltalk [101] notably is strongly message-oriented, and extensions to Smalltalk to a distributed model are common, such as Actalk (Briot [34]), which implements the Actor model [120, 121] on top of Smalltalk 80.

Another notable example is *Erlang* [21, 22, 23], a functional language inspired by Prolog. Erlang was originally created by Ericsson to target embedded systems, but is getting a lot of momentum in recent years to build highly scalable and highly distributed systems.

A defining feature of Erlang as a language is its non-reliance on mutable data; Erlang does not provide shared memory, and only processes immutable data, thereby mapping well onto the dataflow model. Erlang programs are based on having concurrent processes communicating through asynchronous messages (Erlang's creator, Joe Armstrong, calls it a message-oriented language), patterned on Hoare's communicating sequential processes formalism (CSP) [124]. Those features simplify a lot of the programming of highly concurrent and robust systems. As an example, a web server implemented in Erlang (YAWS) still functions well with over 80000 parallel connections, whereas the popular Apache web server dies at about 4000 parallel sessions in the same test conditions [100].

2.2.5.5 Processes Communication Architecture

As the preceding sections clearly showed, one the most critical component of a parallel system is its communication architecture; this is true in hardware, where connections to the memory and remote processors define the performance, as well as in software.

Remote Procedure Calls

In distributed memory architectures (*e.g.* clusters), nodes of the parallel system are interconnected via communication networks (many differerent network topologies can be used [82]). Processes in such systems usually communicates through a form of message passing [41] or via remote procedure calls (RPC) [31].

RPCs are a synchronous mechanism, and as such are not particularly well suited to parallelism, as the calling process have to wait until the called process function returns.

Distributed Objects

An improved approach over RPC is to have distributed objects instead of the simple remote procedure mechanism; a complete object can be manipulated as if it was local, while it is in fact executed on a remote processor, possibly a remote machine.

We should also note that while synchronous call do exist, and are often the default in distributed objects frameworks, many frameworks also propose asynchronous messages, where the calling process continue its execution after sending the message.

An example of this approach is *NeXT Distributed Objects*, implemented by NeXT Computers on their NeXTSTEP operating system. *NeXT Distributed Objects* (DO) leverage the message-oriented nature of the Objective-C language to provide objects able to handle remote messages; conversely a remote object can be integrated seamlessly among local objects. NeXT later rendered DO platform-agnostic as *portable distributed objects* (PDO), which provided an integration with other comparable systems such as CORBA and DCOM.

Common Object Request Broker Architecture (CORBA) [27] provides a common inter-process communication in different operating systems (mostly UNIX-like systems, though there are implementations for Microsoft Windows). Microsoft Windows itself principally uses DCOM as an operating system service. Distributed common object model (DCOM) is an extension of common object model (COM) used to define a standard binary interface so that different languages can create compatible components. DCOM allows different processes to access and share one or more component.

A problem with the above systems is the need to compile and distribute the classes beforehand, possibly on different architectures. Java's Remote Method Invocation (RMI), addresses this by leveraging the machine-agnostic nature of Java.

In all those systems, the unit of distribution is the object itself – objects can be distributed on different computers and interact seamlessly, but a single object is tied to a single computer. In contrast, Globe [229] allows a single object to be distributed across a wide area network.

Data Serialisation Protocols

The type of invocation and model (RPC, Distributed Objects) define the way applications are structured. Another important aspect is how the messages (or invocations) are encoded when transmitted on the network. While historically (with CORBA, DO) the data was encoded using a binary format, with the advance of the World Wide Web, and Web Services, XML is now often used instead. A notable example is the Simple Object Access Protocol (SOAP) [107], used by the Globus grid framework. A simpler and more efficient protocol is XML-RPC [259]. With the introduction of "Ajax" [7] applications, the JSON protocol [61] is another example of a simple, text-based, serialisation protocol.

Futures

Futures [119] are a very interesting model of implicit parallelisation involving messages, bridging synchronous and asynchronous calls. Sending a message to a future object will immediately returns, allowing the calling program to continue its execution. In parallel, the future object can receive the message and execute the corresponding code. When the calling program asks the future object for a value, and only then, will the program block, waiting for the complete execution of the function call (if the result has not already be received). This implicit model allows for either local (the future object executing in a thread) or remote (the future object executed as a distributed object) execution, while implementing the CSP formalism [124].

2.2.6 Parallelisation Patterns

The way systems are architectured, and particularly how tasks are distributed and parallelised, follows commons patterns. In its simplest form, distributing a workload (*i.e.*, parallelising it) consists in broking it into a number of sub-tasks, and aggregating the results.

2.2.6.1 Scatter-Gather and code vectorisation

This simplest form is usually recognised as the Scatter-Gather pattern, the common operation consisting of splitting an original task into sub-tasks (the scatter part), running the sub-tasks in parallel (*e.g.*, on different CPUs or computers), and having the original task aggregating the results (the gather part).

[7] Asynchronous Javascript And XML, a set of web development techniques allowing the creation of more interactive web applications

Historically, major efforts for parallelising code went into automatic code analysis (notably, Fortran [26]), either on multi-processors machines such as the Illiac-IV [218, 154] or later on vector machines such as the Cray-I [197].

When distributed computing took off, the effort switched from code analysis and automatic parallelisation to provide primitives such as scatter-gather mechanisms to the programmer, in popular libraries such as MPI [98]. While those primitives allowed to build scatter-gather applications, they only dealt with the messaging portion of the pattern.

2.2.6.2 Map/Reduce

The Map/Reduce paradigm is a specialised form of Scatter-Gather, coming directly from the functional programming world [228], where the operations are done on maps and keys (*i.e.*, the scattering part *maps* elements and associate keys to them, and the gathering part *reduce* the results by their key).

Map/Reduce [70, 153] is a framework notably used with success by Google that extends this paradigm to parallel computing, formalising the different components and providing a full implementation, taking care automatically of a number of issues (load-balancing, failure recovery, etc.), and as such providing a more powerful paradigm than the simple scatter-gather pattern.

The following quote from [70] clearly summarizes the concept:

> *"The computation takes a set of* input *key/value pairs, and produces a set of* output *key/value pairs. The user of the Map/Reduce library expresses the computation as two functions:* Map *and* Reduce.
>
> Map, *written by the user, takes an input pair and produces a set of* intermediate *key/value pairs. The Map/Reduce library groups together all intermediate values associated with the same intermediate key* I *and passes them to the* Reduce *function.*
>
> *The* Reduce *function, also written by the user, accepts an intermediate key* I *and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output values is produced per* Reduce *invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory."*

The implementation of Map/Reduce described in [70] is as follows:

1. The *Map* invocations are distributed across multiple machines by automatically partitioning the input data into a set of *M splits.*
2. The input splits can be processed in parallel by different machines

3. *Reduce* invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (specified by the user)
4. The process repeats until all map and reduce tasks have been completed.

Map/Reduce implementations

While the original implementation of Map/Reduce by Google is not publically available, other people are starting to implement this paradigm.

Most notably, Hadoop [63] is an Apache Lucene subproject started by Dave Cutting, available as OpenSource, and implementing Map/Reduce on clusters.

QtConcurrent is another recent implementation targeting single multicore computers rather than clusters, provided in the new 4.4 version of the popular Qt C++ framework by Trolltech [246].

2.2.6.3 Pipeline Architecture

Another popular parallel programming paradigm is the pipeline or dataflow architecture [221], notably used in graphics operations. The computation is divided into individual components, organized in a graph structure, with the output of a component going to the input of another component. It is a form of data parallelism, as usually data is passed around to the different components of the pipeline, acting as filters.

Parallelism can be inferred automatically with this architecture by considering the parts of the graph that are isolated and that can be executed in parallel (*i.e.* when two paths of the graph do not depend on each other).

Execution can be driven by the availability of the input data (*i.e.* data-driven) or conversely by the need for specific output data (*i.e.* demand-driven).

The dataflow paradigm has been widely applied in visualisation systems; various existing systems such as OpenDX [5], the Advanced Visual Systems (AVS) [1], IRIS Explorer [4], SCIRun [194, 193] and the Demand Driven Visualizer (DDV) [183] are based on it — organising a visualisation task as a graph of interconnected components.

Theoretically such systems can support dataflow parallelism [227], but as most systems define a coarse-grained dataflow, with most modules not able to handle partial datasets, they in fact offer only limited data parallelism [15].

AVS, IRIS Explorer and OpenDX can all achieve task parallelism with remote modules. SCIRun provides threaded-task and data parallelism on shared-memory multi-

processor. DDV enables a pipeline-based, demand-driven execution that requires the minimum amount of input data to produce the results.

A composite of simple task and data parallelism is stream-based computation, where streams of data are processed by independent components.

Chromium [130] is an example of such an architecture, with streams of OpenGL commands being processed in parallel by pluggable *stream processing units* (SPU). Moreland and Thompson [184] describe a visualisation cluster leveraging the component-oriented nature of VTK [219] and Chromium, by providing parallel rendering components based on Chromium.

Recently, an interesting approach of a graphic pipeline was proposed by Duke *et al.* in [74], where a visualisation pipeline is built on top of a lazy functional language (Haskell). The lazy evaluation capabilities permit better expressiveness when building a pipeline, and allow a demand-driven behaviour.

2.3 Visual Supercomputing

In this section, we introduce the concept of visual supercomputing, and outline its technical scope, from the perspectives of *applications*, *users* and *systems* respectively.

2.3.1 Definition

Visual supercomputing, first defined in [35], addresses the necessary infrastructure for supporting large distributed visualisation systems; we reproduce the original definition below:

> **Definition.** *Visual supercomputing is concerned with the infrastructural technology for supporting visual and interactive computing in general, and visualisation in particular, in complex networked computing environments.*

The infrastructure we put under the label *Visual Supercomputing* not only encompasses hardware technologies, but also software systems, dealing with the computation and the management of the visualisation tasks.

Such software systems have to be able to specify the visualisation tasks — their minimal requirements, performance goals, rendering quality, reliability. They also need to enforce those requirements by managing the task execution: scheduling them on available hardware, configuring the underlying hardware and software systems for the tasks, organising the distribution of the visualised data, the coupling and communication between the visualisation itself and the rest of the system (*e.g.* a simulation application generating data to be visualised, or having the visualisation task wait for live generated data). Furthermore, a visual supercomputing system

should be able to provide support to users' interaction with the system — manage the user experience while accessing and interacting with the visualisation resources.

Visual supercomputing is thus concerned with the broad infrastructure necessary to support such visualisation systems rather than a specific algorithm or technique to be used to process a specific type of data. In fact, visual supercomputing systems should provide support for those algorithms — and choose among them the best approach for a given task.

The closest type of system to such a visual supercomputing environment is the Grid infrastructure (which will be discussed in the next chapter); visual supercomputing can be characterised as a Grid infrastructure for visualisation.

As such it is clear that visual supercomputing is set apart from the traditional domains of visualisation (such as hardware architectures for visualisation, parallel and distributed computation for visualisation, web-based visualisation, rendering algorithms, or collaborative visualisation), as it is more concerned with providing a common environment for integrating all those technologies. Progress in all those areas will obviously impact a visual supercomputing environment, potentially its architecture as they need to be integrated together, but visual supercomputing is ultimately concerned with the infrastructure itself and additional requirements such as the users' experience or the overall quality of services.

2.3.2 Technologies of Visual Supercomputing

Complex visualisation systems, until recently, largely depended on high-performance computing (HPC) environments and infrastructure. Historically, research efforts to bring visualisation capabilities to supercomputers started as early as 1978, with one of the earliest efforts being the vector graphic library for the Cray 1 supercomputer by Elwald and Mass [77].

A huge number of publications describing parallel architectures and algorithms for computer graphics and visualisation have been published since this early work; most of these architectures are no longer used, and many of the algorithms were too hardware-specific to benefit from modern hardware. Nonetheless, this early research brought us a collection of concepts that are still applicable to modern HPC and visual supercomputing environments.

More recent visualisation systems exploit graphics clusters, *i.e.* clusters of machines with commodity graphics cards.

2.3.2.1 Graphics Workstations

While Graphics User Interfaces (GUIs) appeared in the mid to late 1970s, with the Xerox PARC Smalltalk User Interface [101] (figure 2.10, left), they did not so in a vacuum; notably, two important precursors are Sutherland's Ph.D. thesis, Sketchpad [238] in 1963, which had the first window system (as well as being working on graphical "objects", inspiring Alan Kay to create Smalltalk), and Engelbart's oN-Line System (NLS) [79], an hypertext collaboration system presented during a famous 1968 demo, showing the use of the mouse [8].

With graphics workstations, graphics transitionned from being a special feature of a system (*e.g.* with a specific graphics terminal used to output content connected to a mainframe) to an inherent part of the computer's experience. Soon, GUIs left not only the laboratory but even the workstation realm to reach everybody, with personnal computers such as the Macintosh, the Atari ST, the Amiga, and of course the IBM PC with Windows; all GUIs presenting the user with the Windows Icon Menu Pointing device metaphor (WIMP), an interaction model we still experience today.

The workstation's realm continued on its own path, with notable systems such as NeWS and X11 allowing remote display, and powerful workstation such as Silicon Graphics' or NeXT (figure 2.10, right)[9] ; by the late 1980s, individual workstations were powerful enough to allow for interactive 3D visualisation.

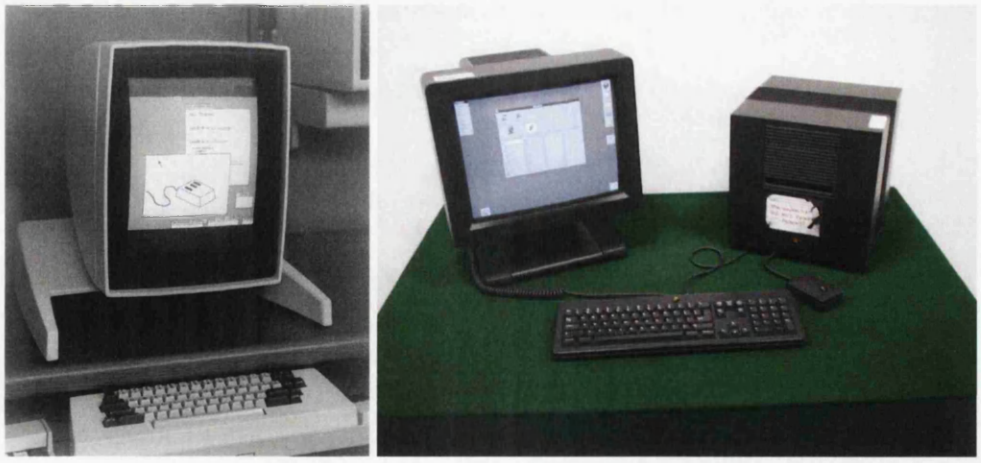


Figure 2.10: *Left, a Xerox Alto (Xerox Parc,* ca. *1979). Right, Tim Berners-Lee NeXT Cube (CERN,* ca. *1988)*

[8] Douglas Engelbart and Bill English are the mouse's inventors

[9] The NeXT Cube was notable for its display PostScript and its object-oriented system, and for being the platform Tim Berners-Lee used to write the first Web server and the first Web browser

Decline of the Workstations

Graphics workstations, powerful but expensive, were gradually replaced with personal computers, starting in the late 1990s, as off-the-shelf graphics cards performances improved.

Companies such as 3Dfx (ironically composed of defectors of the then-leader in graphics workstations, Silicon Graphics), NVIDIA and ATI, started to make commodity graphics cards slowly encompassing many of the high-end features once only found in graphics workstations (geometry engines, etc.).

This decrease in price associated with a broader market created new demands for visualisation tools from users in all types of occupations, for instance security officers or stock-brokers. The availability of this "graphical wealth" also brought into question the role of modern personal computers equipped with powerful graphics hardware in the infrastructure of visual supercomputing.

2.3.2.2 Generalisation of Graphic Hardware

As graphics operations, and visualisation tasks in general, are extremely computationally intensive (working on large amount of data), it is not a surprise that an important and ongoing effort is made toward offloading those tasks onto special-purpose hardware. While during the 1980s, the visualisation field was mostly lead by research and industry-related needs (*e.g.* predominence of graphics workstations such as Silicon Graphics), the democratisation of computers brought a new, insatiable need: computer games. More and more graphically rich games are now the main reasons leading for even more powerful graphic hardware.

Historically, we can cite some important milestones of the development of graphics hardware, such as:

- Video Random-Access Memory (VRAM) [202], which allows dual access (writing to the framebuffer while the digital-to-analog converter (DAC) is reading it), and provides an effective solution to improve the size and access of the framebuffer required by almost every graphics pipeline.
- Graphics processors, such as Intel's i860, led to graphics processing units (GPUs), which first improved 2D performances and later 3D performances (many advances first found in Silicon Graphics workstations such as geometry engines were later adapted to the consumer market, most notably used in 3D games). Ironically one of the first processors used specifically for graphics operations, the Intel i860, was released in 1989 as a general purpose RISC chip, and was Intel's first try of a new instruction set architecture (ISA), departing from the old x86 line. But as performances of the i860 depended heavily on the compiler's ability to fill pipelines, and as runtime code paths were difficult

to predict, the theoretical performances of the i860 were never reached for general-purpose applications. When compiler support improved a few years later, other RISC designs surpassed the i860. Graphic operations on the other hand were able to fit in the cache and it was easy for the compiler to keep pipelines filled; the i860 thus ended up being used as a powerful graphic processor, notably in the NeXT Dimension (32 bits accelerated graphics and video I/O) and in the SGI Onyx Reality Engine 2 (where a number of i860XP processors were used in its geometry engine).

- Multiprocessor graphics architectures, such as Silicon Graphics' POWER IRIS, where the graphics computation was divided in subsystems (geometric manipulation, scan-conversion, visibility determination). The presence of such subsystems allowed to divide the computational cost.
- Texture mapping hardware, allowing pseudo-photorealism environments at a much cheaper cost than detailed geometries. Texture mapping hardware have also been used to develop new visualisation algorithms taking advantage of it, such as texture-based volume rendering [42, 258], flow visualisation [210, 242], splatting [239] and point-based rendering [200].

The flexibility of the latest generations of commodity graphics cards, such as the NVidia GeForce and ATI Radeon families, are allowing more and more applications to take advantage of graphics hardware. Demanding visualisation techniques such as volume rendering and ray casting have already been successfully implemented [174, 209, 213], possibly arranged in cluster [185].

Among all of the increasingly "general purpose" cards, one stands out as a piece of truly special-purpose hardware: the TeraRecon VolumePro, which delivers high-quality and real-time volume rendering capability [198]. Built upon the results of earlier research [199], the commercial VolumePro card currently available for PCs can deliver up to 30 frames per second for a 512^3 voxel dataset.

2.3.2.3 Graphics Clusters

With recent graphics cards bringing more and more computational power to individual desktops, large memory capabilities, and an increasingly more generic nature (programmable GPUs), they became a source of untapped computational wealth for computer scientists. A lot of recent effort has thus been spent on trying to move heavy computational task to those cheap, powerful and ubiquitous cards. Of course, the possibilities are limited by the individual characteristics of the card (such as the amount of dedicated graphics memory, the number of cores, etc.). It is only natural then to try building graphics clusters to build more powerful visualisation systems out of those off-the-shelf graphics cards [185, 270].

One of the first example of such graphics cluster system was WireGL [129], which

allowed the virtualisation of multiple graphics accelerators, providing a sort-first rendering infrastructure. A notable aspect of WireGL is that it presents to the programmer the familiar OpenGL API [187].

The WireGL architecture later evolved into Chromium [130], a stream-oriented system, streaming OpenGL commands to distributed components (streap processing units, or SPU) implemented as a pipeline. It can support sort-first, sort-last and hybrid parallelisation, using the modular capabilities of SPUs.

Chromium has also been used as "visualisation backend" for popular visualisation environments such as VTK [184]. Bethel *et al.* for example combined Chromium with OpenRM Scene Graph [10], a pipelined-parallel scene graph interface for graphics data management.

2.3.3 Design Methods for Parallel Visualisation

Adapting a computation to take advantage of parallel systems such as supercomputers or HPC systems means finding a method to divide this computation into subtasks that can then be distributed onto the parallel system.

In the case of a visualisation system, the computation task to distribute is, given a dataset, the generation of a visual representation of this dataset (*i.e.* rendering an image of the dataset). As expressed by the goals of visual supercomputing, there are obvious reasons to distribute the image rendering, such as taking advantage of multiple displays (display walls), multiple remote clients (mobile computing), and of course to speed up the rendering time if too impracticable for a single computer (large image resolution wanted, complex rendering algorithms used, large datasets to visualise).

Classification of Rendering Parallelisation Methods

Molnar *et al.* [181] proposed a widely used classification of parallel rendering methods in software and hardware rendering systems, considering parallel rendering as a sorting problem.

Figure 2.11 on the following page shows a simplified graphic pipeline adapted for parallel rendering; the two principal parts of this pipeline are the geometry processing step (transformation, clipping, lighting, etc.) and the rasterization (scan-conversion, shading). In such a system, parallelising the geometry processing is done by having each node deal with a subset of the scene dataset, while parallelising the rasterization is done by handing to each node the responsibility for a portion of the pixel calculations. A rendering computation is at its core a matter of sorting the primitives on the screen; in the case of a parallel system it means to sort and distribute the primitives on the processing nodes. Molnar *et al.* thus postulate that

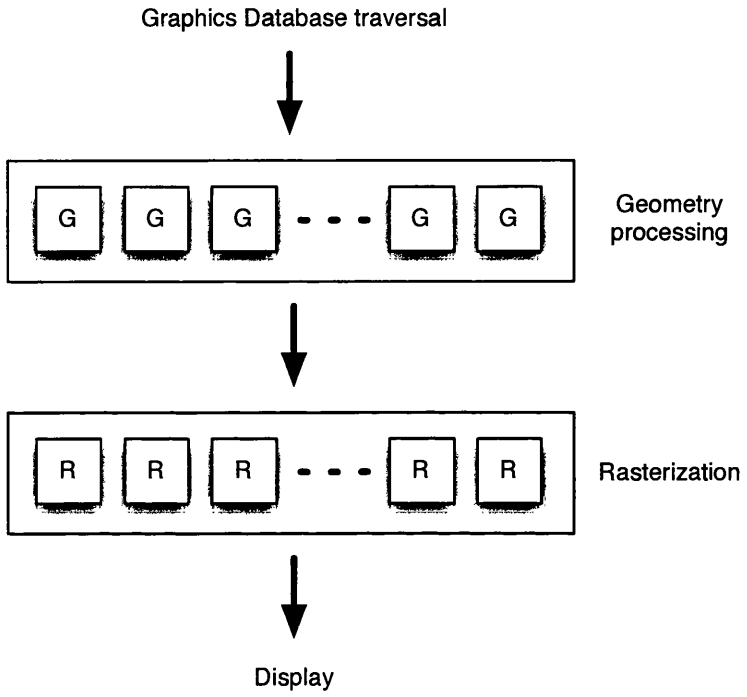


Figure 2.11: *Graphics pipeline in a parallel rendering system. Processors* G *perform geometry processing. Processors* R *perform rasterization [181]*

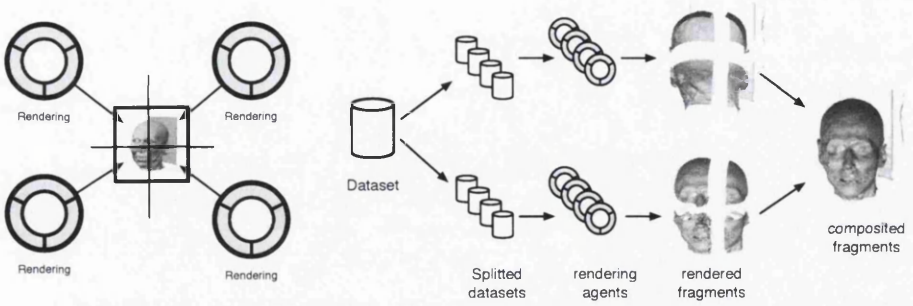
the sort can take place at any position in the graphic pipeline shown on figure 2.11, and propose the following classification :

- *Sort First*, before the geometry processing, *i.e.* redistributing the raw scene primitives before their screen-space parameters are known
- *Sort Middle*, between the geometry processing and the rasterization, *i.e.* redistributing the screen-space primitives
- *Sort Last*, after the rasterization process, *i.e.* redistributing the pixels

Image-Space and Object-Space Rendering

Two common categories of rendering parallelisation are *Image Space* and *Object Space* [104]. Both approaches are forms of data parallelism, creating subsets that can be distributed on a parallel system and used to generate partial visualisations that are later merged in a final representation. Both approaches presuppose the existence of a *graphic pipeline*, where the original dataset is processed to finally generate an image.

Image Space parallel rendering (figure 2.12 on the following page, left) considers the final wanted result as a 2D image; this 2D space is divided into sub-areas that can



(a) Image Space *(b) Object Space*

Figure 2.12: *Image Space and Object Space Distributed Rendering*

be rendered independently, in parallel, by different nodes. The final image is created by compositing the partial images together.

Conversely, *Object Space* parallel rendering (figure 2.12, right) considers the original dataset first, and splits it into smaller datasets that can be processed in parallel by different nodes. Those nodes generate full or partial images using those partial datasets. The final image is created by compositing the partial results together, taking in account their organisation into the object space. This method is useful when the data size is too large to be processed easily by a single node, and the dataset has to be split into more manageable subsets.

Image space parallel algorithms can be identified as the sort-first approach, as the sub-images are chosen before any geometry processing, while *object space* parallel algorithms can be identified as the the sort-last method, as it is necessary to sort the resulting images and merge them together depending on the objects' position to create the final result.

Image space parallel systems allow an easy parallelisation, and when the data to render is not a bottleneck (*i.e.* when it is possible to easily distribute or share it). It works particularly well for a setup such as display walls; but Image space rendering have a limited speedup as the scene is duplicated on each node, which can quickly become a limiting factor for large scenes.

Object space systems allow a better speedup (as larger scenes can be processed), but have an even more expensive recomposition step than image space systems, which can itself limit the actual rendering speed.

2.3.3.1 Image Composition

Image composition is the last stage of the graphic pipeline, when all the partial results have to be integrated to a final output. It is a known bottleneck in distributed rendering systems [191], as intermediate steps in the rendering processes

are distributed on many computers, but typically the integration is done on a single, final node. This bottleneck can potentially seriously limit the scalability of a distributed rendering architecture, and is an important aspect of a distributed rendering pipeline.

Different projects proposed hardware accelerated compositors to great success, such as Lightning-2 [231] , Sepia-2 [164, 118, 180] or SAGE [71].

Using such dedicated hardware increase performance, but as with any hardware solution they are less convenient to integrate with existing systems and are less common than off-the-shelf hardware. Software composition mechanisms have been proposed such as [157, 260, 232]. As our own system will work on a cluster of machines without any specific hardware allowing for image composition, we will have to implement a specific image composition architecture in software.

2.3.3.2 Data Distribution

Data distribution and partitioning is of particular importance for distributed visualisation systems; for image space rendering systems the data need to be distributed to (or be accessible from) all the rendering nodes, while for object space rendering systems the corresponding subdatasets need to be sent to their assigned rendering nodes.

Two important requirements of a good data distribution are data locality and load balance. Achieving data locality implies that the communication overhead stays low – it is better for a rendering node to access the data locally than it would be to access it remotely. Load balance tries to ensure that the computation load is divided as equally as possible among the rendering node. Load balance is particularly important to achieve good efficiency in parallel systems.

Partitioning algorithms often try to exploit data or spatial coherence to ensure data locality, while minimizing the amount of data residing on each node. Mackerras and Corrie [167] for example exploit data coherence to improve parallel volume rendering.

Data Replication

Data replication is thus an important aspect of a parallel rendering system. The type of replication can usually be classified in the following forms: complete replication, block replication, and structured replication. With complete replication, each node will have a copy of the rendered data; while good for achieving great speedup and load balancing, this method obviously has limited scaling possibilities. Block replication divide the data in different parts, either regularly or irregularly (which can be helpful to achieve a good load balancing). Finally, structured replication

organise the data in a hierarchical partitioning, which can help the data distribution. A good example of such approach is octree subdivision [73, 54], which recursively divide the object-space into height octants. An octree can be used to organize the dataset according to various attributes, such as the spatial occupancy or the workload [253]. Similar commonly used plane partitioning structures are *BSP tree* [96] and *quadtree* [217]. While most of these structured partitionings take place in the object space, many of these methods can also be adapted to image space parallelisation, as they can facilitate efficient view-dependent data fetch [162], as well as combined image and data coherence. In recent years, scene graphs were used as a hierarchical structure for managing sort-first, distributed memory parallel visualisation [29], and helping with real-time virtual reality applications [186].

Load Balancing

As stated previously, parallel visualisation systems need to prevent as much as possible load imbalance; *load balancing* is generally addressed when assigning tasks to the rendering nodes, by trying to balance the average workload of a node. The different types of load balancing are usually classified by their run-time behaviour:

- Static Task Assignment [260, 166] computes the workload of each rendering node, depending on the predicted workload of each subtask and on the available processing power of the rendering nodes. While this approach requires some pre-processing before assigning the tasks to the nodes, it implies less communication overhead at runtime. Data coherence is usually taken into account during the task assignment, allowing more efficient data partitioning and distribution.
- Dynamic Task Assignment (*e.g.* [151]) needs to maintain a pool of tasks (usually small, to simplify the load balancing). When a rendering node becomes available, it receives a task from the pool, and the system iterates until all the tasks in the pool are processed. This batch processing approach is effective in heterogeneous environments (where each node can have varied computation capability) and image space parallelisation (as the load of each subtask is hard to predict).

2.3.4 World Wide Web and Distributed Visualisation

The World Wide Web allows users to easily navigate frome one document to the other, by following links; the geographical dimension is abstracted by a common network protocol (HTTP) and a global addressage system (URIs), and is not generally relevant. Not only documents allow links toward other documents, they can also embed resources, referred through the same addressage mechanism; those resources can themselves be located anywhere.

This hypertext nature made the web easy to use and pervasive; it is now the de facto standard for publishing documents, and web technologies have even be stretched much further than the original document orientation. Powerful applications can now be created as web applications, rendering the host operating system even more irrelevant.

2.3.4.1 Visualisation on the Web

All those characteristics make the idea of Distributed Visualisation based on a web platform appealing. Ang *et al.* were the first to propose an architecture allowing visualisation to take advantage of the web capabilities. Their idea was to allow the distribution of 3D volume datasets through the normal web mechanism, using a specific mime type labelling the data as a volume dataset; the web infrastructure only intervened as a way of distributing the data easily, and possibly to allow the easy creation of specific website to browse the datasets.

Of course, simply serving a resource would not work as the browser would not know what to do with it; they thus wrote a plugin for a web browser [10] that would talk with a local application in charge with the rendering. When a user would go to a page embedding a volume dataset, the plugin took charge of the rendering, asking the local application to generate an image. The local application then required available rendering computers, which would be harnessed to render the image in parallel. Once done, the image would be pushed back to the browser's display.

While a complete architecture, we can see that the web was only used as a communication and publication method.

Client-Side Rendering

A similar approach was followed by Michaels and Bailey with VizWiz [177]. The custom distributed parallel rendering application and the plugin system were replaced by Java applets; the complete rendering was then done by the applet. One possibility offered by VizWiz was to upload a local dataset onto the VizWiz server, thereby allowing a (limited) form of collaboration. This Java applet approach did not encounteer much success, which was likely caused by the dramatically poor performances of Java VM at the time.

[10] One of the first popular web browser, NCSA Mosaic

Server-Side Rendering

Logically, the other approach integrating web and visualisation is to have the server rendering the image instead of the client, and transmit the result.

A slightly different possibility, implemented for example by Wood *et al.* [264] and Engel *et al.* [78], is to use the server to transcode the original dataset into a format more suitable (in their example, VRML [28]) to the client's characteristics (*e.g.* scale it down).

Server-Side rendering is likely to stay the main architecture, as it allow much larger scaling up compared to rendering done exclusively on a client machine (as the rendering capabilities would be limited by the client machine's capabilities).

Another aspect to consider is the increasing power of client-side technologies, which may impact this question (*e.g.* fast javascript VMs, native access, etc.).

2.3.4.2 Collaborative Distributed Visualisation

Distributed visualisation allows users to view complex datasets. Such datasets can comes from various origins (medical data, results of simulations, etc.). At its root, visualisation allows people to observe a specific dataset and analyse it. The need for organising a collaborative visualisation process is tied to the complexity of the examined data, and the large datasets that are the target of distributed visualisation and visual supercomputing are prime examples of this need.

Internet, as a global publication network, allow people to easily publish documents, and thus encourage collaboration. The Internet infrastructure itself provide key mechanisms that can be reused in for a collaborative distributed visualisation.

If we consider collaborative visualisation, we need to define what is shared (to allow for the collaboration to take place), and how. The simplest possible way of sharing a visualisation is to share the final display; another way would be to share the visualised data, with each user rendering it locally. The ideal system would allow to freely mix elements coming from different users in a single pipeline, so that each users could choose which parts of the rendering pipeline is imported from other users, or exported to them.

Display Sharing

Display sharing is implemented by conferencing technologies (*e.g.* Microsoft NetMeeting), but screen sharing applications such as VNC [8] are widely used. Different display sharing protocols exist, the most widely used being VNC (cross-platform, open), RDP (Microsoft, proprietary), ICA (Citrix, proprietary), X (open), and NX

(cross-platform and able to encapsulate other procotols). The simplest method to share the display is the one used by VNC, where the entire screen is regularly captured, and a stream of compressed bitmaps representing screen updates is sent to the remote client. More advanced protocols such as Microsoft's RDP, X or NX protocols can work at the drawing primitives level, allowing lower latency and smaller update streams.

Data Sharing

Data sharing has been exploited in collaborative environments such as CUMULVS [3] and in pV3 [7], where data from parallel computations is made available to multiple viewers. Another example of data sharing is provided by COVISE [2] where geometry is made accessible to a group, each person in the group being able to render as they please.

Full Collaboration

The idea of having a full collaboration process is to share elements of the rendering pipeline with other participants; those elements can be the data (exported or imported in the pipeline), or parameters (isosurface, camera viewpoint). Wood *et al.* [263] demonstrate this approach with the Covisa project [263], an extension to IRIS Explorer.

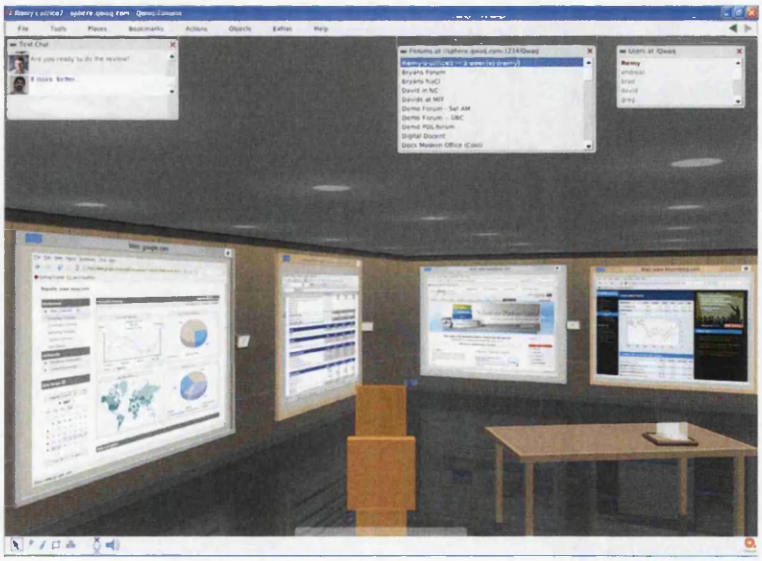


Figure 2.13: *Qwaq Forum is based on OpenCroquet and provides a virtual meeting space to business users [143, 204]*

Virtual Environments

More recently, virtual environments (Figure 2.13 on the preceding page) such as Second Life [161] or OpenCroquet [143] allow collaboration through a full 3D environment. While Second Life, a commercial product, has more impact (social, political and cultural events happen routinely), Croquet has a stronger educational focus, as well as being open (it is based on Squeak Smalltalk [136]), and provides a good distributed platform to connect different 3D worlds.

Many important issues control the architecture of collaborative visualisation system; technical issues such as system heterogeneity, computation capacity imbalance, network bandwidth imbalance (*e.g.* DSL connections), but social issues such as security, privacy or floor control have important roles.

2.3.4.3 Impact of the Internet on Visualisation Systems

As we described in the previous sections, both web-based and collaborative visualisation systems already present examples of a visual supercomputing infrastructure. The web is already one of the dominant information highways, and as such it is very likely that a visual supercomputing infrastructure will have to provide a substantial amount of its services through it.

2.4 Applications of Visual Supercomputing

We presented in the previous sections the different components involved with visual supercomputing; notably, the available hardware architectures and the common programming paradigms, how distributed computing relates to visual supercomputing. While a new paradigm, visual supercomputing is not a solution looking for a problem; in [35], we presented several areas, such as mission critical visualisation, visual data mining or large-scale visualisation, which could benefit from a visual supercomputing infrastructure. We present here two potential areas related to the system we developed, computational steering and mobile visualisation.

2.4.1 Scientific Computation and Computational Steering

Scientific computation, generating a large amount of data, is one of the area where the visual supercomputing paradigm will have the bigger impact. One of the key aspect of scientific modelling and simulation is computational steering, where systems are organised around a feedback loop — parameters and data generating results, which can be visualised and analysed [248]. Users can then steer the computation by changing the parameters and observing the results.

Considering the problem of using visualisation in simulation environments, Marshall *et al.* [170] identified three possible ways of combining them: post-processing, tracking, and steering.

With post-processing, visualisation is simply used as a final step, not involved with the simulation part – the visualisation cannot be used to modify the simulation while it is running.

Tracking implies that the visualisation runs in parallel with the simulation, allowing the user to observe it in real time; but the simulation cannot be modified.

Finally, with steering, the user can observe the simulation results in real-time, modify the simulation's parameters and immediately see the results of its actions reflected by the visualisation part.

Brodlie *et al.* [37] proposed an extension of the model to add a log of checkpointed information, stored in an history tree. These checkpoints can then be used by the user to go back in the simulation session if needed.

Various systems provide steering capabilities. An example of a steering environment is SCIRun [193], which provides a dataflow environment designed for computational steering. Another approach is to use a framework such CUMULVS [3] to link simulations with steering and visualisation services.

Recent efforts such as the RealityGrid project [39] implement steering systems as Grid applications.

Computational steering is notable because of the tight coupling between the computation (the simulation) and the visualisation. This demonstrates the need to implement advanced inter-process and inter-components communication and management in a visual supercomputing system.

One aspect of scientific simulation is that it often amounts to a repetitive feedback loop, where the same data is explored, or the same simulation is run using different parameters. This would allow a visual supercomputing system to gather performance data associated with additional information (such as the parameters used); a good visual supercomputing architecture ought to be able to use this gathered information to help the user (*e.g.* automatically choosing the best parameters).

An example of such automatic parameters optimisation was presented by Wright *et al.* [269], in the related area of design steering. Using numerical optimisation techniques, they automatically try to maximise the performance of a particular design by exploring the design parameters' space. Such automatic optimisation though is highly dependent on the parameters space and on the existence of an evaluation function.

2.4.2 Mobile Visualisation

Now more than ever, mobile devices are parts of our everyday's life. Not only are they pervasive, but their capabilities greatly increased: recent mobile devices are rather similar to desktop computers from a few years ago — some even based on unix systems (iPhone, Linux-based devices, Android).

Integrating those mobile devices into the visualisation pipeline seems a straightforward possibility to extend the reach of visualisation applications, and allow users to access the pipeline remotely, possibly interrogating or manipulating the visualised data.

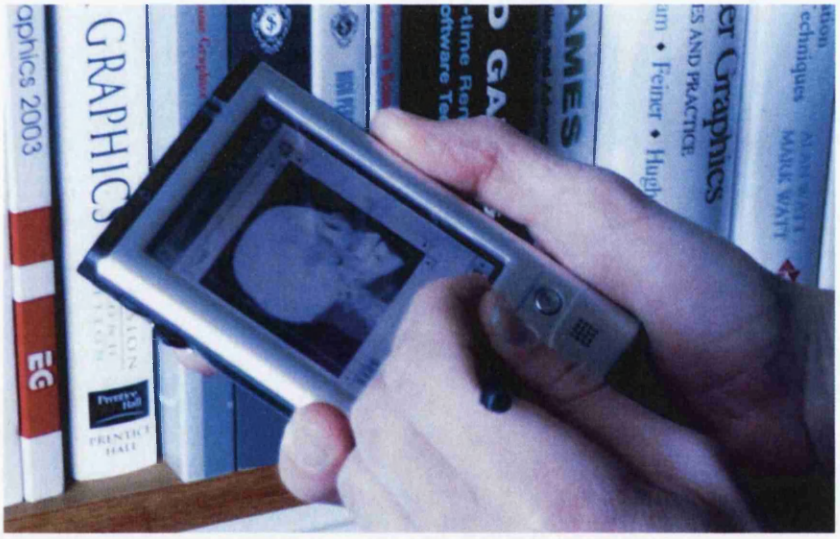


Figure 2.14: *Mobile technology has offered an exciting scope for developing new visualisation application (M. W. Jones, Swansea University)*

Collaboration Aspect

Integrating mobile devices in a collaborative environments has been proposed by Izadi *et al.* [137], with the FUSE system, a Jini-based [236] platform to support ubiquitous collaboration for multiple users, allowing intermittent communication and roaming devices.

FUSE is inspired by Weiser's ubiquitous computing vision [256], where a multitude of computers are available throughout the environment, while being mostly invisible to the user — a vision that seems more and more within reach, as mobile device presence increases.

Mobile Devices as Clients

The general approach to transform mobile devices into visualisation tools is to make them clients of an existing rendering pipeline; the mobile device will simply act as a remote display, with the rendering done via a remote server or cluster. More rarely is the device itself doing the rendering, due to the imbalance between the mobile device's rendering capabilities and a remote rendering server or cluster.

Using Personal Digital Assistants (PDAs) as visualisation tools was demonstrated by Lamberti *et al.* [152]. Their system is based on a remote rendering Chromium cluster, with a software bridge sending the rendered images to the PDA. The PDA is not just a remote display, but allow users to explore the rendered scene via an ad-hoc navigation interface.

Another example of using the PDA as a remote control is Tweek [115], a middleware system presenting users with a Java GUI, connected to a C++ Virtual Reality application through Corba. Tweek displays a 2D map of the VR environment to the user, allowing them to better interact with it.

D'Amora and Bernardini [67] implemented an interesting client/server architecture allowing a PDA to visualise VRML CAD models. The CAD models are stored on the server, but instead of doing the rendering on the server, the models are transcoded in an optimised format and then transmitted to the PDA. A custom 3D viewer then load the models and display them on the PDA. The advantage of this model is that the PDA is autonomous and the interaction is potentially better (being rendered locally); but the original models are degraded due to the PDA limitations.

Role of the Mobile Device

We can use the communication requirements to classify how mobile devices are integrated in a visualisation environments: remote scheduling, remote monitoring, remote steering, and remote visualisation.

In remote scheduling, the device is only used to monitor the status of a remote visualisation task, possibly allowing for basic actions such as starting, stopping, holding or removing the task. Graphics are not necessary, the entire interaction could happen via a simple text interface or even message-based.

Remote monitoring is similar to remote scheduling, with the added possibility of checking the visualisation state, *e.g.* getting a recent rendered frame from the visualisaton task; minimal graphic capabilities are thus necessary.

Remote steering adds the possibility to change visualisation parameters, controlling from time to time the results as with remote monitoring; this implies larger communication bandwidth, possibly low latency.

Finally, remote visualisation involves a full visualisation process, usually in real-time, where the user can freely control what is displayed. This of course put the most constraints, as a large bandwidth and low latency are necessary for a successful visualisation process.

Even so, remote visualisation of course regroups under a single term widely different performances, depending on the available bandwidth, and the device capabilities. Recent mobile devices provides accelerated 3D chipsets, with high-density screens.

An interesting approach by Wolf *et al.* [261] is the Smart Pointer concept. Instead of having the mobile device acts as a simple remote display, similar to a normal client apart from the display size, users are presented with a subset of the visualised data. The mobile device or PDA can then also act as a remote control for a larger display.

Mobile visualisation (figure 2.14 on page 47) can be seen in different ways; one possibility is to think of mobile devices as underpowered computers with small display areas, and adapt the visualisation services to those constraints. Another aspect is to concentrate on the communication limits (available bandwidth, latency) to overcome for a successful visualisation process.

While concentrating on those constraints is necessary, specifically if we want to integrate mobile devices in a visualisation process (as visualisation is highly impacted by those), the more interesting aspect of mobile devices is not to be forgotten; that is, their mobile nature, necessitating the development of specific interaction models or UI [195].

In addition to this, recent devices can now be used as if "always on", *i.e.* always connected to the Internet, or having the possibility to be. Another recent, yet more and more common capability, is the ability to determine the user's current location.

All those characteristics make mobile devices challenging, but potentially extremely rewarding platforms.

2.5 Conclusion

We presented in this chapter a thorough review of the related concepts and technologies visual supercomputing is based on.

We first reviewed parallel systems: taxonomies, models of parallel computation, hardware architectures.

We introduced the concept of system modelling to evaluate and predict performances. Among the presented models, parametric models have the advantages of being simple and fast, with a reasonable accuracy, and more importantly do not need a huge amount of data. This makes them ideal candidates for real-time systems such as the one presented in this thesis, and we will present some results in Chapter 8 showing the use of those models in our system.

We described common parallel programming models, and common distribution mechanisms such as scatter-gather and Map/Reduce. Map/Reduce is notable in reducing the amount of specialised code for doing task distribution, allowing the programmers to concentrate on the tasks rather than the mechanisms. We will provide an implementation of this mechanism in our system to add a higher-level approach to parallelising code. A notable application of this paradigm in our system will be the implementation of an efficient distributed rendering pipeline.

The second part of this chapter introduced and defined the notion of *visual supercomputing*, followed by a review of the technologies used in visualisation, and by extension, in visual supercomputing, in hardware and software.

Image rendering parallelisation methods, Image-Space and Object-Space rendering, were presented. Our rendering pipeline will notably implement an Image-Space rendering method. Two typical problems were also highlighted: the image composition bottleneck and the load balancing need.

Finally, we described a few potential use cases where visual supercomputing can have a strong impact, notably mission critical visualisation and mobile visualisation.

(CHAPTER ...3)

Managing Complex Systems

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable"
— Leslie Lamport (CACM, June 1992)

Contents

COMPLEX DISTRIBUTED SYSTEMS involve many different resources, processes, networks and data. Different approaches were developed to reduce and manage the inherent complexity of such systems. We will first outline the different problems found in such systems, and present two research areas related to this problem: the Grid computing effort and the autonomic computing initiative. We will discuss how these technologies relate to the problem of visual supercomputing, and introduce the e-Viz project.

3.1 Distributed Systems

Distributed Systems are systems composed of a collection of computers connected and working together. Many different definitions exist of distributed systems; we will here use the one put forward by Tanenbaum and van Steen:

Definition. *A distributed system is a collection of independent computers that appears to its users as a single coherent system [241]*

3.1.1 Advantages of Distributed Systems

The principal goal of a distributed system is to share and pool together resources. Sharing a few costly resources among many nodes induces a better efficiency, as the resources are more likely to be used more often than they would be if isolated.

Pooling resources (notably computational resources) allows users to work on problems that would be too big for a single computer, or at a cheaper cost (*i.e.* an equivalent single computer would be more expensive or simply impossible to build).

With an application running on a distributed system, it is easier to achieve good reliability, as single component failures only impact a fraction of the whole system.

Lastly, applications running on distributed systems can scale better, simply by adding more nodes to an existing system.

3.1.2 Disadvantages of Distributed Systems

While greater application reliability is one of the potential advantages of a distributed system, individual component failures happen more often than on a monolithic system, simply because the number of interactions and the number of components involved in a distributed system is greater. On large distributed systems, the question is not *if* a component will fail, but *when*.

Failures in distributed systems can be of many different forms; algorithmic and architectural problems (such as deadlock issues, bandwidth bottlenecks, bad distribution of the data), software bugs (unexpected crashes), or even hardware (computer nodes failing, or unreachable).

Managing failures is one of the important aspects of a distributed system that we will cover in the following sections.

Another problem of distributed systems is that algorithms can sometimes be difficult to adapt to a distributed approach. Furthermore, porting applications to a distributed system can be a tedious task compared to the (relatively) simple task of writing an application for a single computer system, due to the added complexity.

More fundamentally, the efficiency of a distributed system will never reach 100% percent, as resources are wasted due to synchronisation and communication among the nodes. The efficiency of a distributed system (usually measured as the ratio of

the *speedup* [139] to the number of nodes) is an important metric, and depends on the software and the hardware architecture.

Lastly, managing a distributed system can be a complex task on its own, increasing with the size of the distributed system, the number of applications and failures to cope with. Recent efforts such as the Grid [91] dedicate an important part of their infrastructure to this aspect.

3.1.3 Complex Distributed Systems

A typical example of distributed systems is a cluster of machines arranged as a single system. Such clusters are usually homogeneous (each node being identical in terms of software and hardware).

We will introduce here a slightly different concept, which we will call *complex distributed systems*:

> **Definition.** *A complex distributed system is a distributed system built on top of heterogeneous components: software, hardware and networks.*

Two notable examples of such complex distributed system are the Internet, and Grid systems (themselves using the infrastructure created by the Internet).

In addition to sharing the same potential issues as homogeneous distributed systems, a large part of such complex distributed systems has to deal with accommodating the differences of their components, and how to use them as efficiently as possible.

3.2 System Failures and Fault Tolerance

Although reliability concerns should be of particular importance in a distributed system, as it is by default a less robust architecture than a single computer system (*e.g.* communication failures rate being much higher in a distributed system than with a single computer system), it is by no means a unique property of such systems; reliability concerns exist as well in non-distributed systems, the distributed nature only acting as a magnifier.

We can define a system fault as such:

> **Definition.** *A system fault occurs when an unplanned, often incoherent state is reached by a running system.*

A fault-tolerant system is one that can still provide the service it is in charge of even in the case of system faults; the concept of such systems is not a novelty (W.H. Pierce wrote a book in 1965 about *Fault-Tolerant Computer Design* [201]). In 1975, Randell introduced the notion of *software fault tolerance* [205] , in opposition to the hardware-only fault tolerance techniques mostly used until then. Different techniques are

described, notably *recovery blocks*. Conceptually close to database transactions [103], such software blocks identify sections of a program where errors can be detected and recovered after restoring the original program's state.

3.2.1 Distributed Computing Fallacies

In a distributed system, there are a number of known failure points — notably, communication failures and synchronisation issues, as well as possible unreliability of computer nodes used by the system. A characteristic of a distributed system is that the more nodes are used in a system, the more the chances of a failure will increase.

Designing a distributed system thus needs to take into account those potential failures – the question one should ask is not *if* a failure will occur, but *when*. It is easy to overlook some of the design constraints of such systems. L. Peter Deutsch (a fellow of Sun Microsystems at the time) identifed in the 1990s seven *Fallacies of Distributed Computing* [72, 125, 214] a list of the bad design assumptions one can have about a system; James Gosling later added the eighth one.

1	The network is rellable
2	Latency is zero
3	Bandwidth is infinite
4	The network is secure
5	Topology doesn't change
6	There is one administrator
7	Transport cost is zero
8	The network is homogeneous

Table 3.1: *Fallacies of distributed computing [72]*

Table 3.1 lists those fallacies, which give an excellent summary of potential problems found in a distributed system. Those problems are still highly relevant, even though some of the metrics listed (latency, bandwidth...) have improved dramatically since then — as with computational power, expectations rose faster than existing solutions. The difficulty for system designers is to work around those issues, by presenting to users a stable system even in case of failures or dramatic changes. It

is often instructive to compare system characteristics to this list, and understand which part of the list the system addresses, and how.

3.2.2 Categories of Fault Tolerance Techniques

In [206] Randell *et al.* identify four basic categories of techniques used to provide fault tolerance in computer systems:

- *Error detection.* The system tries to recognize possible errors just before they occur, in order to limit error propagation.
- *Fault treatment.* When a component of the system has been deemed faulty, a proper course of action needs to be taken; it usually triggers replacing strategies (the component is replaced by an equivalent one) or reconfiguration strategies (the system may be reconfigured to keep working without the component, or with settings avoiding the error).
- *Damage assessment.* If an error occurred, the system might be corrupted, and the amount of damage needs to be evaluated.
- *Error recovery.* In order to recover from an error, the system needs to return to a correct state. There are different possible approaches to this, usually involving atomic operations or transactions.

We will use this classification in the following sections to describe different techniques used toward fault-tolerance.

3.2.3 Error Detection

The oldest method to detect errors is simply to check the return value of the executed block of code, mixing the error detection with the code itself.

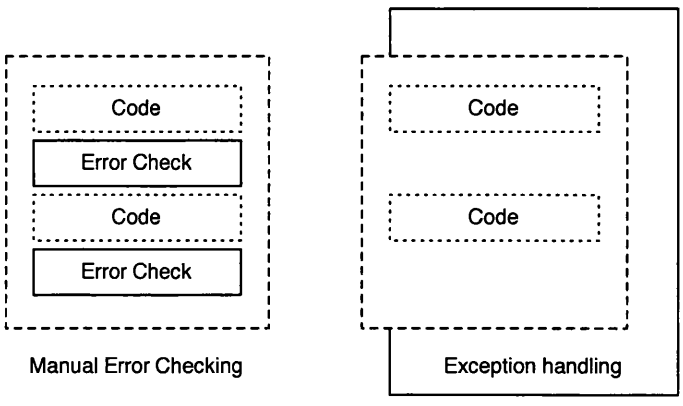


Figure 3.1: *Manual error handling* vs *exception handling [102]*

Nowadays, probably the most common way to detect errors and limit their propagation is the use of exceptions in programming languages. Figure 3.1 on the previous page shows both models.

Goodenough formalised exception handling in programming languages in 1975 [102], giving the following definition:

> **Definition.** *Of the conditions detected while attempting to perform some operation,* exception conditions *are those brought to the attention of the operation's invoker. The invoker is then permitted (or required) to respond to the condition. Bringing an exception condition to an invoker's attention is called* raising an exception. *The invoker's response is called* handling *the exception.*

The exception mechanism is notably different from previous error handling mechanisms in that the operation causing the error is not permitted to act on the error. Treating an exception must be done outside the software block causing it. This allows a cleaner separation of concerns in programs, allowing programmers to create different exception handling strategies to resume operations.

One of the first languages implementing exception handling as a built-in mechanism was CLU, from Liskov and Snyder in 1979 [163]. Other languages such as Ada [135, 134] in 1979, or Smalltalk [101] in 1981 quickly followed, and provide built-in support for exceptions. Most programming languages in current use now propose this functionality in one form or another.

3.2.3.1 Error Detection in Distributed Systems

Distributed systems are formed of many collaborating software components. A common class of error occurs when one or more than one component becomes inaccessible — due to a network error, a crash of the components, or even a hardware failure. Detecting such faults is thus an important part of making a distributed system reliable.

Failure detectors are components in the distributed system that detect failures and notify other components when a fault occurs [116].

In distributed systems where messages cannot be delivered within a known time, Fischer *et. al.* [85] proved that consensus is impossible if one component is faulty, as faulty components cannot be distinguished from merely slow ones.

As highlighted by Chandra and Toueg [50], failure detectors can be used to overcome this impossibility result for consensus in asynchronous system models [85, 249].

Failure detectors are implemented using the *push* and *pull* models [80, 116].

With the push model (Figure 3.2 on the following page), components periodically send hearbeat messages to a *passive* monitoring component (also called a *failure*

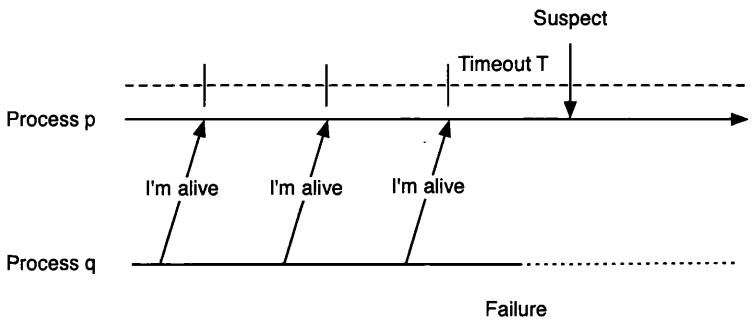


Figure 3.2: *The push model [116]*

detector); when the monitor does not receive the hearbeat after a time interval T (timeout), it can flag the component as having failed. This model is characterised by a potentially unbounded number of messages — with large systems, these could potentially flood the network.

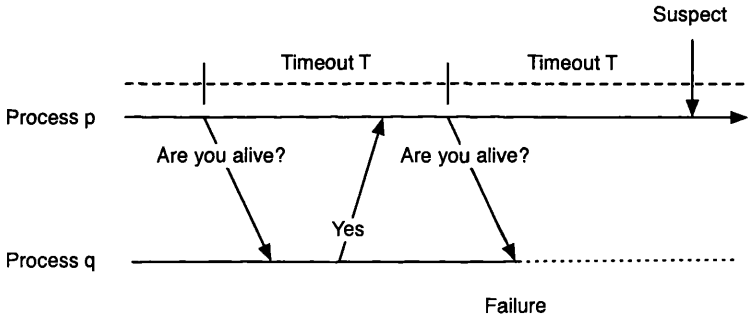


Figure 3.3: *The pull model [116]*

In reverse, the pull model (Figure 3.3) has the failure detector being *active*, sending "are you alive" messages to the surveyed components. The components reply; failure to do so has the components flagged as being unavailable. With the pull model, the network load is possibly reduced and more importantly controlled by the monitor (as components will only send back the messages as answers to the monitor queries). On the other hand, the monitor component will only detect the failure after it decides to ask the components if they are alive.

3.2.3.2 Hierarchical Failure Detectors

Felber *et al.* [80] introduce the concept of hierarchical failure detectors, where failure detectors are monitoring components, and reporting conditions changes to clients components.

Figure 3.4 on the following page shows how failure detection could happen with the following situation: three components m_1, m_2, m_3 are respectively monitored

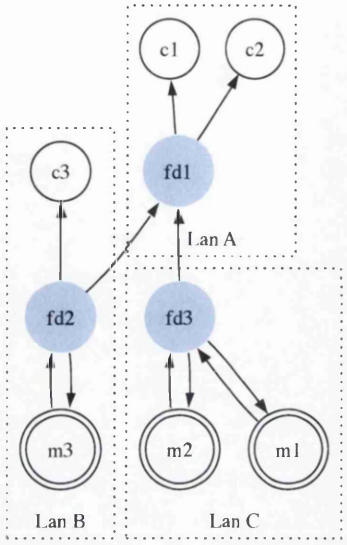


Figure 3.4: *Hierarchical failure detectors [80]*

by three components c_1, c_2 and c_3, but all those components are not localised in the same local network. In a classic organisation, we could have a failure detector component per LAN; this would increase the number of messages exchanged, possibly dramatically if the system is composed of a large number of components. Organised hierarchically, each failure detector fd_1, fd_2, fd_3 only monitors local components, but *optionally* forward notifications about monitored components to other interested failure detectors living in *different networks*. In this example, the components c_1 and c_2 are interested in notifications about m_1, m_2 and m_3, hosted on different networks. Their local failure detector fd_1 thus asks the corresponding failure detectors in charge of m_1,m_2,m_3 to forward notifications regarding their status. This setup limits the potential message explosion when dealing with systems spanning different networks.

It is worth mentioning that the basic concept of failure detectors — detecting failure in remote components using timeouts — was previously (and successfully) implemented in the network realm; a notable example being the TCP protocol [49] which provides reliable communication over IP. The internet control message protocol (ICMP [203]) is another example of a network protocol using the same concepts to check failure in remote machines, using *echo* packets to check if the remote machine is reachable [1].

[1] "Echo Request" and "Echo Reply" packets are used. The ubiquitous *ping* network utility sends such ICMP requests.

3.2.3.3 Gossip-style Failure Detectors

Gossip-style failure detectors [208, 223, 207] implement gossip protocol to broadcast failure detections. Renesse *et al.* proposes two types of protocols, a basic one and a multilevel one.

The basic protocol has a failure detector on each host, maintaining a list of other failure detectors known to itself and a heartbeat counter for the surveyed components. Regularly, the failure detectors send to a randomly chosen detector their list, which is merged.

The multilevel gossping protocol follows an internet-like domain decomposition, with messages aggregated before being sent to another subnetwork. The gossip approach works well to tackle the message explosion problem, and can adapt to system topology changes. On the other hand, this type of protocol is less efficient than hierarchical ones.

3.2.4 Byzantine Errors

The previous section on error detection is focused on detecting errors happening during the communication process — *i.e.* the components involved are considered to be either working and reachable, or not reachable, independently of whether they are still running or not.

Another type of potential error in a distributed system happens when a component sends conflicting information in the system, possibly on purpose, modelled after the Byzantine Generals' problem as presented by Lamport *et al.* [155].

In other words, in the previous section we consider the components as perfect while the communication channels are not. Byzantine errors consider the reverse problem: we consider the communication channels as perfect, while the components are not.

The general approach to treat Byzantine errors is to reach a consensus among components even though some of them can be faulty. Different algorithms were proposed to overcome this problem [155, 81], such as quorum-based protocols [11], or asynchronous state-machine replications [47, 48].

While treating Byzantine errors is a theoretically important issue in distributed computing, particularly at the scale of the Internet where rogue components inserted in a system are potentially a much more likely problem, we will not discuss this issue further: within the problem of visual supercomputing treated in this thesis, we consider that our experimental visualisation system is isolated and components are considered as safe.

If we had to securise our system, we may keep a similar architecture (*i.e.* where the internals of a system — machines on the local network – are considered safe) as a

first approach, with encrypted/signed channels between remote systems (a solution usually taken by Grid systems).

3.2.5 Fault Treatment

When a fault occurs, the system can either completely stop (*i.e.*, crash, in the worst case), provoking a default of service, or try to resume its operation. Ideally, a system will try to resolve the root cause of the fault, and then retry the operation that triggered the fault.

In distributed systems (or component-based system in general), a few actions are possible to handle a fault, as presented by Hwang and Kesselman [131]:

- *retrying* the component
- *checkpointing*
- *replication*
- *replication with checkpoint*

Hwang and Kesselman provide a comparative analysis of these four principal failure treatment strategies. They conclude that for environments with high downtime, replication with checkpointing performs the best; on the other hand, in environments with low downtime, checkpointing techniques (*i.e.* checkpoint and replication with checkpoint) have a lower completion delay.

Replication of components associated with regular checkpointing thus appears as a good strategy for distributed visualisation systems, where low downtime should be expected. An important aspect that a system should offer is thus an easy way to checkpoint the current state.

Component Replication

Restarting components or invalidating them in case of faults can be quite costly, as the new components might have to be reinitialized and reintegrated with the system. A general solution is to have multiple instances of a component running (although possibly not in use, but ready to act) in parallel, which will reduce the latency between a fault and the system resuming its operations. Duplication strategies are a vast research topic by themselves [165, 252], and are usually system-dependent. As a viability mechanism, a pure duplication may not be effective as software flaws would repeat themselves, possibly at the same time. Avionics systems typically address this concern by using three independent implementations of the same system in parallel, and using a simple voting mechanism to determine the correct action to take (*i.e.*, two out of three subsystems need to agree to trigger an action).

3.2.6 Damage Assessment and Error Recovery

One of the main difficulties with dealing with exceptions and faults is to architect a system so that it can fall back on a valid state whenever a fault occurs [205]. A well-known strategy is to use atomic operations, which will either entirely pass or entirely fail, which simplify error handling (the system will not end in an unknown state). But atomic operations are a subset of transactions [103], a concept particularly used in database systems, which allow to run several operations as if they were atomic — the transaction will either fail or pass. One difficulty with transactions is that the different operations usually need to be idempotents, as the transaction system needs to possibly rollback those operations.

Another way to provide rollback mechanisms is to implement checkpoints [76] — the system will regularly save its state, which will allow it to revert to the last saved (working) state when a fault occurs. Rollback implemented with basic checkpoints can suffer from the *domino* effect [215]; as an instantaneous global checkpoint is not feasible, there may be an unbounded cascade of rollbacks in order to find a consistent view of the system (if checkpoints are too coarse). Log-based rollback [233] can remove this domino effect. The system logs all the operations, synchronizes and replays the operations to restore the state.

Transaction mechanisms allow a system to recover gracefully from a failed series of operations, without ending in an unknown state.

A completely different model with error recovery is the REST model (see Fielding Ph.D. thesis [83]), where state is passed to components explicitly with each request, with each component being stateless. If a component fails, it is enough to restart it and pass it the request, without needing a full system reconfiguration. Of course this only takes care of hardware failures and possibly software errors in the component, not software errors in the system itself. This is a particularly popular model in web applications and web services.

3.2.7 Failure and Reliability in Distributed Systems

Distributed systems tend to have failures occuring more often, by the simple fact that the more computer nodes they use, the more likely a fault will occur. As such, they can be considered as being inherently less reliable than single system architectures; in fact the reverse is true.

While failure rates are indeed higher with a distributed system, they can be structured to remove single points of failure, and allow duplication of the system components. If a fault occurs, the system performance might be degraded, but the system should be able to resume its operations transparently by using redundant components. This is in fact one of the most interesting aspects of a distributed system, as

counting on the lower failure rate of a single computer system is merely pushing away the next catastrophic failure.

A particularly well-known example of the advantages of a distributed system in terms of reliability is the Internet — Internet servers are hidden behind an indirection level (their domain name). To reach a server, one has to know its IP address, which is returned by DNS servers [179] [2]. It is usual to have more than a single server answering requests for a given domain name. As requests are stateless with HTTP, this architecture can easily scale up. More importantly, the architecture is completely transparent to the end users.

3.3 Evaluation of Parallel Systems

The previous sections discussed distributed systems architectures and problems, notably the failure detection mechanisms. An important issue with distributed systems and parallel systems in general is how we can compare and evaluate systems, in view of the variety of hardware architecture and software middleware used.

We will present in this section different metrics commonly used in parallel systems evaluation.

3.3.1 Evaluation Techniques

There are many different techniques that can be used when measuring the performances of a parallel system [149, 142]. An important metric is the completion time of a given job; but this alone is meaningless when evaluating the system's merit on its own — an efficient system running on a cluster composed of slow processors could be vastly outperformed by an otherwise inefficient system running on a cluster blessed with fast processors; additional metrics are therefore necessary.

The primary objective of using N processors in parallel to solve a problem of size M is the multiplication of the amount of processing power, commonly measured in terms of MIPS (*millions of instructions per second*) or FLOPS (*floating-point operations per second*). However, as previously outlined, it is not possible to parallelise all problems perfectly without introducing additional costs.

One widely used performance metric is the *speedup* [16, 139], the ratio of the time taken by the fastest-known sequential algorithm to that of a given parallel algorithm executed on N processors. In theory speedup can never exeed the number of processors N [16], though in practice *super-linear speedup* (speedup > N) may

[2] The DNS architecture is itself one of the biggest existing distributed systems, managing millions of records, and taking advantage of their hierarchical organisation to scale particularly well

sometimes be observed [113] due to the effects of a particular system architecture. As Gustafson demonstrated in [113], one way of obtaining super-linear speedup is to scale the problem's size with the system's size — at constant problem's size the system would present sub-linear speedup, but scaling up the system would also allow bigger problems to be explored.

It is also important to measure a parallel system's *efficiency*, which is defined as the ratio of speedup to the number of processors; and the *cost* metric, which is the product of parallel run time and the number of processors used. One design goal for a parallel algorithm is to achieve a *cost-optimal* system, the *cost* of which is proportional to the execution time of the fastest-known sequential algorithm. The main obstacle to achieving a *cost-optimal* parallel system is the *overhead* resulting from parallelisation, which is usually caused by inter-process communication, extra computation (e.g. initialization, distributed data management), and idle waiting (e.g. load imbalance, task synchronization).

Increasing the number of processors reduces efficiency, while increasing the size of the computation increases total speedup, hence efficiency. Another important metric is *scalability* [147], the capability of a parallel system to maintain efficiency by increasing problem size and speedup in proportion to the number of processors.

Time-constrained scalability [112] is the core issue in some applications, such as weather forecasting, where it is necessary to fix the parallel run time, and to scale the problem size according to the number of available processors. Gustafson *et al.* [112] also examined the *memory-constrained scalability*, focusing on the largest problem that can fit the available memory in a parallel system.

3.3.2 Notable Metrics

We will present here some of those metrics that present a particular interest in allowing experimental evaluation of systems.

Speedup

The speedup of a system is one of the defining metrics used to evaluate a parallel system. We presented Amdahl's law in the precedent chapter, and will simply here present again the formula shown section 2.6 on page 18, considering its importance in evaluating systems experimentally, as well as being used in the definition of common metrics. The speedup S is expressed as $S(N)$ for N processors:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{3.1}$$

Efficiency

Another important metric is the *efficiency*, which measures how close to an ideal usage of the available resources the system is. The efficiency is usually defined by the following equation, where S is the system speedup and N the number of processors:

$$e = \frac{S}{N} \tag{3.2}$$

A system with low efficiency might not be automatically discarded as such, as the actual cost of the resources used may be less. A typical example is shared-memory architectures such as the SGI Origin 2000, versus low-cost clusters built with off-the-shelf computers; the efficiency of the cluster will very likely be less than the shared-memory machine, as communication links (speed, latency and bandwidth) are definitely less competitive, but the cluster may end up being better as more cost-effective to build, compensating low efficiency with more computers.

Serial Fraction

A less known but particularly useful measure of parallel performance is the serial fraction of a parallel system, introduced by Karp and Flatt in [142], with S the system speedup and N the number of processors:

$$f = \frac{1/S - 1/N}{1 - 1/N} \tag{3.3}$$

Where the smaller f is, the better the parallelisation. This metric is also known as the Karp-Flatt metric.

The evolution of this metric as processors are added is particularly informative, as Amdahl's law is an overly optimistic law, which considers that the amount of work per processor is equally divided.

In reality, load balancing a task is a complex problem, and f captures this; load imbalance will be shown in f as irregular changes while N increases. f also shows the overhead (*e.g.*, synchronisation) introduced by adding more processors. Lastly, one big advantage of using f is that it is constant — in theory — which makes it easier to detect variations in real-life systems.

These characteristics make f a particularly useful diagnostic tool when analysing and comparing parallel systems.

3.4 Grid Computing

Grid Computing [89, 91] is an example of a Complex Distributed Systems (as presented in section 3.1.3 on page 53); the main differences with classical cluster computing are indeed its heterogeneous nature (machines composing the grid can vary widely in terms of computational power and architecture), the fact that individual elements of a grid can be geographically dispersed, and as a consequence, loosely coupled. In contrast, a typical cluster uses similar machines, connected via a fast local network; we could define grid systems as a way to share computer power and storage over the Internet.

The principal advantage of the grid concept is the possibility to build large computational power using commodity hardware; another advantage is the possibility to integrate as part of a grid specific resources (*e.g.* supercomputers) that can then be easily shared by the users of a grid. Universities or institutions can for example team up to build a common computational grid (*e.g.* the UK National Grid Service).

3.4.1 Public-Resource Computing

The disadvantage of a grid is, unsurprisingly, the loose coupling and slow data link compared to a supercomputer or even a local cluster. As such, the grid approach tends to be better suited to computation that can be executed in parallel, where each part of the computation can be done in isolation. For such problems, the amount of computational power that a grid infrastructure can provide can scales very well: historically, a popular application of the grid concepts has been shown with `distributed.net` (1997) [9] and SETI@Home (1999) [257, 18], allowing hundred of thousands of people to donate spare CPU cycles to those research grids (this approach is also called public-resource computing, to illustrate the contrast with more organisational grid setups). The amount of computational power gathered that way is far from ridiculous: SETI@home provided a sustained processing rate of over 70 TeraFLOPS in 2004 [17], and the Berkeley Open Infrastructure for Network Computing (BOINC [17]) announced in early 2008 that it provided a sustained processing power of 1.06 PetaFLOPS (BOINC is the infrastructure used by SETI@home and now used by several other research projects).

3.4.2 The Globus Toolkit

The nature of the grid, assembling such varied types of computers, architectures and networks to work together, mandates a software infrastructure. Grid middleware is thus necessary to interconnect all those varied systems as a single, virtual system that applications can use.

One of the most popular middleware (notably in institutional grids) is the Globus toolkit [89, 90, 93]. Globus is based on open standards and Web services, which helped with its adoption. It is developed by the Globus Alliance, an international association dedicated to developing technologies used by grid computing, and is an implementation of the standards developed at the Open Grid Forum (OGF), notably the Open Grid Services Architecture (OGSA).

OGSA [92] describes a set of capabilities a grid have to implement:

- Infrastructure services
- Execution Management services
- Data services
- Resource Management services
- Security services
- Self-management services
- Information services

Those capabilities are implemented on top of various existing Web services technologies. The Simple Object Access Protocol (SOAP) [107], a protocol for exchanging structured information using an XML syntax, forms the base of the message exchange in Globus. Globus also needs to be able to work with components and services having a state; Web services are stateless, so Globus implements since version 4 of the toolkit [3]. the Web Services Resource Framework (WSRF) [64], which provides a set of interfaces that a Web service have to implement to become stateful.

3.4.3 Visual Supercomputing and The Grid

Visualisation on the Grid [220] is a challenge, due to the nature of it: slow data links (compared to a cluster), loose architecture, high latency. Different strategies (doing the entire rendering on the server and sending back frames to clients, rendering the data on the clients) need to be chosen dynamically to fit changing requirements.

Existing architectures such as the one described by Norton and Rockwood in [190] push the rendering on the clients, while the data is managed by the Grid. While this approach allows visualisation of data generated from a grid service, it does not answer the problems posed by Visual Supercomputing, where the rendering process itself is too complex to be executed on a single client.

Another grid-enabled architecture is RAVE [105, 106], which uses the Web Services Description Language (WSDL [57]) to discover rendering resources automatically. The rendering is based on Java3D [235] and can be either client-based or server-based, depending on the capabilities of the client machine; yet, the server-based rendering though is not distributed.

[3] Previous versions of Globus were using the Open Grid Services Infrastructure (OGSI [247])

Recent work from Ahmed *et al.* [14] implements a grid-based pipeline, where the individual elements of the pipeline are on different machines, following the dataflow decomposition from Haber and McNabb [114], but while each step (data reader, isosurface extractor, VTK mapper and VTK render) is on a different machine, the rendering step is not further distributed.

3.4.4 Failure Detectors in Grid Implementations

The Globus toolkit provides a failure detection service [230], using two layers of failure detectors. The lower layer provides local monitors that survey their local host and selected processes running on that host. The upper layer consists of data collectors, receiving regular updates from the local monitors. A problem with this architecture is the lack of dynamism, not reacting well to changes in the topology.

Felber *et al.* [80] propose hierarchical failure detectors, as exposed in the previous section presenting failure detectors. However their approach, being based on a static tree, also lacks dynamism.

3.4.5 System Prediction in Grid Systems

In recent years, much work has been done on adding system prediction to Grid systems (*e.g.* [141, 40, 234, 95]), with a strong focus on resource management (a key problem in heterogeneous environments such as Grid systems).

Predicting performance is also logically becoming a research topic in autonomic systems [60, 168, 145, 271]. Crawford and Dan proposed eModel [60], a Java framework where users implement different classes (data collection, event monitoring/handling, workload modelling) and integrate the framework into their system. Mancini *et al.* [168] describe a simulation framework, MAWeS, to support development of self-optimizing predictive autonomic systems for Web service architectures. It is based on MetaPL [171], an XML-based language to describe distributed systems and able to be used for performance analysis, and HeSSE [172], a simulation tool allowing the user to simulate performance behaviour, using traces obtained through application instrumentation.

Parametric models — such as Amdahl's law — are particularly interesting for evaluating complex dynamic systems, as they can be used without needing a large amount of detailed information and adaptation to the particularities of the system; their simplicity makes them great candidate for real-time systems.

An example of this approach was recently demonstrated by Fu and Huang [95], with a service-oriented Grid monitoring system using a simple forecasting algorithm based on k-Moving Average $MA(k)$ and Exponential Smoothing Model $ExS(\alpha)$.

3.5 Autonomic Computing

Autonomic computing [144] is a research focus initiated by IBM in a late 2001 manifesto [126]. IBM observed that as computer systems becomes more and more complex, there will be a point where it will not be possible to correctly configure, optimise, or fix those systems by human intervention, if only by lack of people. The solution is to automate systems' management and configuration, so that bigger systems will be possible, and the created complexity manageable. Their proposed approach to implement this is to model the systems as being autonomous — in reference to the way the human body works. Systems should therefore be able to "run themselves", adjusting to existing conditions, optimising themselves to be as efficient as possible, with human intervention possibly limited to setting goals.

The manifesto posit eight characteristics such an autonomous system should have:

1. have detailed informations about itself
2. configure and reconfigure itself under varying and unpredictable conditions
3. tries to optimise itself continuously
4. able to recover from routine and extraordinary events causing a malfunction
5. able to protect itself to attacks
6. knows its environment and adapts to it
7. need to follow open standards to communicate with other systems
8. anticipate the amount of resources it needs

Kephart and Chess [144] point that the essence of autonomic computing systems is *self-management*, and list four properties to address it:

- Self-Configuration
- Self-Optimisation
- Self-Healing
- Self-Protection

3.5.1 Self-Configuration

Current systems are complex to install, configure and integrate with each others. This complexity can also result in errors, even from experts. Autonomic systems should be able to configure themselves automatically, as well as integrate themselves in an existing ecosystem. Ideally, autonomic systems will follow a set of high-level goals set by an administrator.

Melcher and Mitchell [175] discuss the problem of an autonomic network service configuration, whereas existing protocols such as DHCP do not provide all the flexibility they want; they propose implementing an autonomic network configuration using the IBM Autonomic Computing Toolkit [132].

Cheng *et al.* [53] propose a coordination architecture to support automatic composition of self-management components. They point the potential interference risk in having independent self-management components, as they could try acting on the same parameters.

3.5.2 Self-Optimisation

Any large software system also have a large number of parameters, and tuning a system for optimal performances can thus be a tedious and complex task, tied to evolving requirements. In addition, performances are also linked to the system's environment, which can change quickly; a self-optimising autonomic system should thus be able to act on dramatic changes in the environment. An ideal self-optimising system will thus constantly try to find the most optimal configuration, possibly using past performances as a guide.

Walsh *et al.* [254] implemented utility functions in Unity [244], an autonomic system based on agents and using OGSA, to demonstrate self-optimisation. Using utility functions, they were able to select among competing system objectives.

Kirtane and Martin [145] propose to use performance prediction for Autonomic systems, describing a performance model for transaction-oriented systems.

Yoo *et al.* [271] propose four different approaches (ID3, Fuzzy Logic, Fuzzy Neural Network and Bayesian Network) to be used as prediction models for self-healing systems, using historical data.

3.5.3 Self-Healing

As we discussed in previous sections of this chapter, error recovery is a large problem. A system should be able to react upon error, either software or hardware, in an independent manner, and recover gracefully. Systems could use internal and external knowledge (sharing informations with other systems) to identify possible causes and solutions.

Most existing autonomic system (*e.g.* [244, 13]) monitor components and restart them upon failure.

3.5.4 Self-Protection

Modern systems not only have to deal with hardware and software failure, they also must have some protection against directed attacks or cascading failures (such as denial of service). Security and validation should be a concern for autonomous systems, possibly analysing real-time data to anticipate problems.

3.5.5 The MAPE-K Autonomic Loop

A reference model for autonomic control loop was proposed by IBM [133], often called the Monitor, Analyse, Plan, Execute, Knowledge (MAPE-K) Loop.

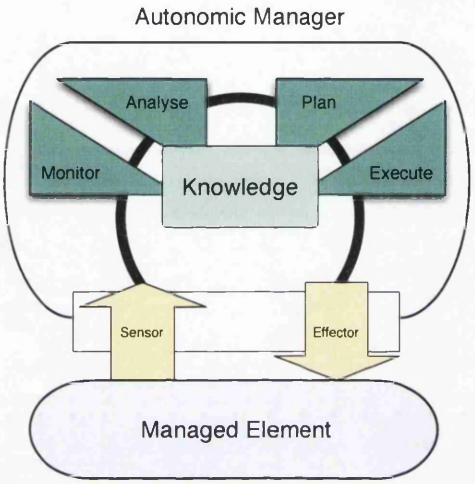


Figure 3.5: *MAPE-K Loop*

As shown on figure 3.5, in the MAPE-K loop, an Autonomic Manager receives data about a monitored (managed) element via sensors, and can modify the element via effectors. The manager is composed of four parts arranged in a loop, respectively monitoring the element (using the sensors), analysing the gathered data, planning an answer, and executing it through effectors; each part can contribute to the internal knowledge database or use it to make decisions. This model is often used to classify different autonomic systems, in terms of which part of the loop they cover.

Similarities With Software Agents

Autonomic systems are somewhat related to Multi-Agents Systems; one of the goal of autonomic computing is to have elements working together toward a commong goal [133], a fundamental aspect of multiagent systems. This also explains the similarity between the generic autonomic manager described by the MAPE-K loop and the generic knowledge-based agent model as described by Russel and Norvig [216].

It is therefore quite unsurprising to find many implementation of autonomic systems based or partly based on software agents. As an example, Automate [13] is a framework for enabling autonomic Grid applications that makes use of software agents to implement the system's rules [159, 158, 160]. The system itself is divided

in different layers implemented using OGSA services, implementing a System, Components and Application separation.

3.6 The e-Viz Project

The e-Viz project [35, 211, 38, 36] was started in January 2005, and explored the infrastructural technology needed for visual supercomputing. E-Viz was a collaboration between four UK Universitites: Bangor, Leeds, Manchester and Swansea.

This Ph.D. thesis was funded through an EPSRC grant allocated to the e-Viz project, and present work performed toward the goals pursued by e-Viz.

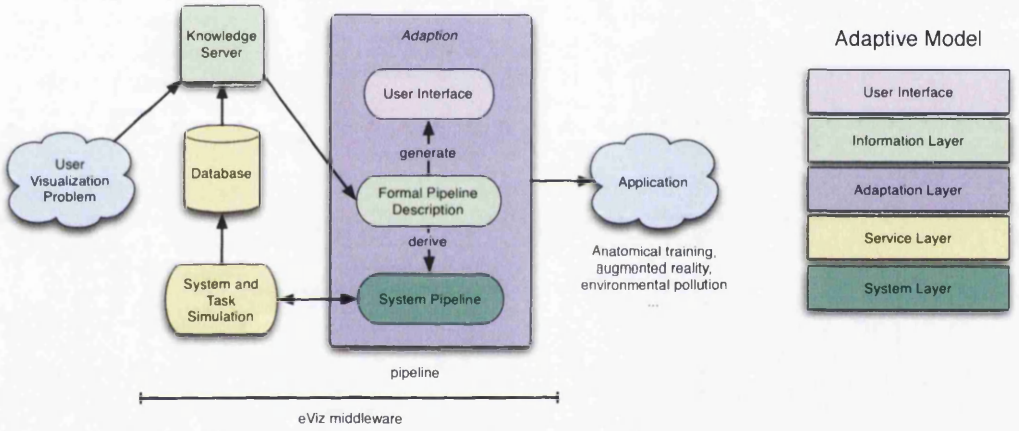


Figure 3.6: *e-Viz architecture*

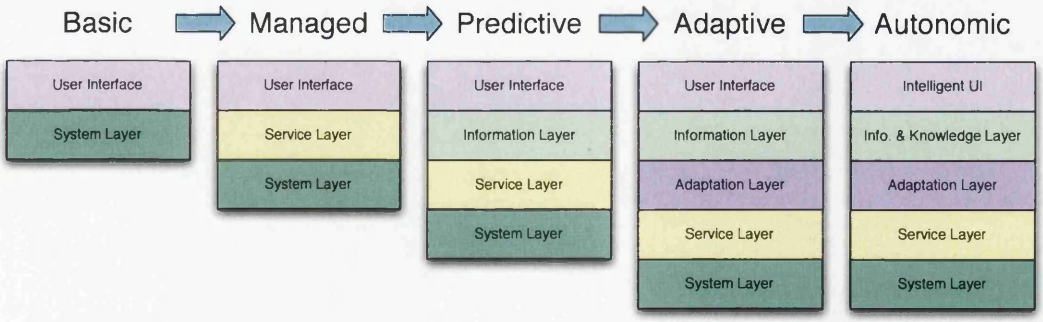


Figure 3.7: *e-Viz layers evolution model*

The general e-Viz architecture is shown on Figure 3.6. E-Viz followed a layered architecture (Figure 3.6) that allowed it to evolve gradually toward an autonomic architecture.

The general mechanism consists to having the knowledge server use information

from the service layer to create a graphic pipeline description. This formal description (derived from gViz [262]) allows the creation of a user interface. The adaptation layer uses information from the bottom layers to adapt the pipeline to changing runtime conditions. The knowledge server makes use of SimuVis [55] to predict future performances.

While the complete solution for the problem of visual supercomputing is well beyond the scope of a single Ph.D., this thesis aims to present a consistent agent-oriented framework allowing to implement such a system. This thesis does not cover work done by other members of the e-Viz project (Table 3.2 on the following page shows the general areas each university concentrated on); notably out-of-core rendering work by Chisnall *et al.* [56] would be of particular interest for a general Visual Supercomputing architecture. Work on reactive user interfaces was done at Leeds and Manchester [38], and work on using the adaptive grid infrastructure for augmented reality at Bangor [128, 127].

3.7 Conclusion

We presented in this chapter an overview of distributed systems concepts, introduced the notion of system reliability and detailed the different categories of reactions a system can implement to provide a safeguard for failures. Reliability in distributed systems is a primordial notion, and we will implements features such as failure detectors in our system.

We then described different popular performance-related metrics used to evaluate parallel systems. Most notably, metrics such as the efficiency, the serial fraction and of course the speedup of a system will be used to evaluate our system.

We reviewed Grid computing and autonomous computing, two research areas closely related to our efforts.

The Grid is a distributed computing infrastructure dealing with heterogeneous and distant resources, and providing services needed by modern requirements (authentication and security, QoS). When working with complex systems as defined in the beginning of this chapter, management and optimisation of the system can become in itself an intractable problem. Autonomic computing aims to make systems more autonomous, by modelling them like the human body, using autonomous components that are able to take decisions. We reviewed in this chapter recent work on this approach. One of the main problem with using the Grid infrastructure for visual supercomputing is the lack of interactivity and the high latency of the infrastructure; our goal with this thesis is to partially bridge the gap between grid concepts and visual supercomputing, using the ideas put forward by autonomous computing.

Finally, we presented the e-Viz project, a visual supercomputing effort which Swansea University participated in, and which this thesis was a contribution to.

Swansea	Volume Visualisation
	Volume Graphics
	Computational Fluid Dynamics
	Mobile Visualisation
	Interactive Visualisation
	Stream-based Visualisation
Leeds	Very Large Data Visualisation
	Distributed and Collaborative Visualisation
	Models for Linking Simulation and Visualisation
	Computational Fluid Dynamics
	Computational Biology
	Environmental Sciences
Manchester	Scientific Visualisation
	Visualisation of Coupled Models
	Virtual Prototyping
	Very Large Data Visualisation
	Computational Steering
	Distributed Visualisation
	Medical Visualisation
Bangor	Mobile Visualisation
	Scientific Visualisation
	Image Based Surgery
	Augmented Reality
	Interactive Visualisation

Table 3.2: *Tasks distribution in the e-Viz project*

PART II

System Design

(CHAPTER ...4)

Design of an Agent System

"But just to show how stubbornly an idea can hang on, all through the seventies and eighties, there were many people who tried to get by with 'Remote Procedure Calls' instead of thinking about objects and messages."

— Alan Kay, 2003

Contents

THIS THESIS aims to explore an autonomous distributed visualisation system. A key design point of our approach is that the system is entirely built on top of a multi-agent system, which provides to the upper levels of the system a set of high-level features: communications between components, management of component pools, failure protection, etc. This chapter will describe the agent system, detailing its architecture and the different implemented mechanisms. Note that we will focus on the general features of this agent system, *i.e.*, non-visualisation related. The following chapters will detail the visualisation aspect of the system.

Before describing the general design approach, and considering that the agent-oriented approach we took is fundamental to the system, it is important to answer some questions:

- What are software agents?
- Why is this approach interesting to us?

4.1 Software Agents

To answer the first question, we can quote the definition given by Wooldridge in [267] :

> **Definition.** *An agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives.*

Jennings [138] gives further explanations; agents are:

- clearly identifiable problem solving entities with well-defined boundaries and interfaces
- situated (embedded) in a particular environment – they receive inputs related to the state of their environment through sensors and they act on the environment through effectors;
- designed to fulfill a specific purpose – they have particular objectives (goals) to achieve;
- autonomous – they have control both over their internal state and over their own behaviour;
- capable of exhibiting flexible problem solving behaviour in pursuit of their design objectives – they need to be both reactive (able to respond in a timely fashion to changes that occur in their environment) and pro-active (able to act in anticipation of future goals) [268]

4.1.1 Language Perspective

Software agents are directly inspired from the *Actors model* introduced by Hewitt in 1973 [120, 121]. Agents are independent and autonomous actors collaborating together.

Of particular interest to us is Kay's Smalltalk [101] programming languages, one of the first object-oriented language. An important notion of Smalltalk put forward by Kay is that every object acts like a model of a computer: the system is self-referential.

In Smalltalk, objects communicate by sending messages. Conceptually, the fact that Smalltalk usually runs on a single virtual machine is purely an implementation

detail — the model allows easily to integrate remote objects transparently [66].

An example of adapting Smalltalk to the actor model is Actalk (Briot [34]), which implements the model as a minimal extension of Smalltalk 80.

Smalltalk can be seen as a close conceptual model of a collaborative agent system, with Smalltalk objects posing as agents and collaborating by exchanging messages. The Smalltalk paradigm of messaging also allows it to be easily adapted to distributed computing. As such, Smalltalk can be seen both as an inspiration for software agents and as a good modelling platform, possibly an implementation platform, for software agents.

4.1.2 Agent Communications

The principal difference between software agents and "normal" processes running concurrently, apart from their autonomous nature, is the existence in agent systems of a common language among agents [1]. Agents can send messages to other agents to express intentions or needs or, to generalize, the messages exchanged have well-defined purposes.

Such languages (usually called ACL, for Agent Communication Languages, see [84, 245, 150]) allow agents to collaborate by sharing the same understanding about the possible messages. Agent systems have to provide such a common communication protocol to their agents.

4.1.3 Software Agents and Autonomic Computing

The intrinsic autonomous nature of software agents helps answer our design goal – to create an autonomous system. Jennings [138] put forward the notion that an agent-oriented approach has the potential to significantly improve our ability to model, design and build complex, distributed software systems. The main reason for this is that there is a high degree of match between the requirements of complex system development – decomposition, abstraction and organisation as identified by Booch [33] – and agent-based computing.

It is therefore not surprising to find agent-based architecture often used in autonomic computing efforts; projects such as Unity [244] are notable examples, and many other projects follow a similar architecture [178, 140, 273, 108].

It is worth pointing that autonomic efforts efforts do not exclusively use software agents — some research only looks into specific aspects of the autonomic problem, as we presented in Chapter 3. Complete autonomic *systems* that are not agent-based isolate specific software components whereupon autonomic features are applied (*e.g.*

[1] Such common language can be seen as a specialised communication protocol among agents.

AutoMate, [13, 192]). One way of considering this is to make an analogy between such efforts and functional programming *vs* object-oriented programming; with OOP, objects are responsible for themselves, which we could equate to agents, while in functional programming we apply functions onto data, which we could equate to such autonomic efforts applying rules onto software components.

Both approaches are valid and can lead to interesting results in autonomic terms. In addition, applying rules to components allow a better integration of such autonomic features in existing systems.

Nonetheless, using a software agent approach is both a more integrated and a more componentised approach, which we believe is a superior way of implementing autonomic features and explains why many existing autonomic projects also follow that path. It is also our case that we developed a system from the ground up, without being tied to legacy code, which made the choice of using a full agent system easy.

The Unity project [244] is a good example of such agent systems oriented toward autonomic features. It is composed of different components (*autonomic elements*, which are individual software agents) controlling resources and delivering services to users or to other autonomic elements. Every Unity component is an autonomic element, responsible for its own behaviour, such as managing the resources it needs, configuring and optimising itself. Separate *application environments* can be supported by the system, with each environment supporting a specific application.

Architecturally, each application environment is represented by an *application manager* agent; notable other types of agents are *resource arbiter* (in charge of the resource allocation to application environments), *servers* (representing the resources), *registry* (implementing the naming mechanism), *policy repository* (reifying high-level policies guiding the system operation) and *sentinel* agents (acting as failure detectors). Technically, Unity uses a web services standard such as OGSA [92] and is implemented in Java using the Autonomic Manager Toolset [24].

4.1.4 A Proposed Architecture for an Autonomic System

The previous sections presented the concept of software agents and their characteristics. As postulated by Jennings [138], this particular software architecture maps well on the construction of complex systems. As such, software agents are a good candidate to the realisation of an autonomic system, as the Unity project other similar efforts can demonstrate.

We first introduced an early version of our system in [212], which presented some examples of reflective capabilities [25, 146] to create an autonomous system on top of an agent system. While both this early version and our current implementation follows the same general design principles, they vary widely in terms of implementation; we reimplemented our own low-level communication protocols instead

of using XML-RPC [259], introduced broadcast mechanisms into the system, and removed the centralized parts of the system to have a fully distributed system not depending on any fixed point. In addition to these low-level changes, a lot of new high-level mechanisms were also added: relationships, the map/reduce paradigm, and more reflective capabilities.

In the following sections we will detail those differences and describe the various mechanisms forming our system. The design approach we followed is illustrated by table 4.1, which shows the different layers of abstraction composing the system.

7	Autonomic System
6	Intelligent Applications
5	Reflective System
4	Graphic Pipeline
3	Software Agents
2	Distributed System
1	Processes

Table 4.1: *System Layers*

If we describe each layer, we have processes (*1*) interacting in a distributed system (*2*) by sending messages. On top of this distributed system we build a multi-agents system (*3*) – each process is in fact an agent, which provides to the next layers a range of functionalities (creation, discovery, etc.). Using those agents, we implement a distributed graphic pipeline (*4*). Agents, interacting in a common environment, can feed on shared information, in essence creating a reflective system (*5*). Using the information of this reflective layer and the autonomic nature of agents, we can create intelligent applications (*6*), which will culminate in a full autonomic system (*7*).

As in the Unity project, a unifying approach is followed, whereby all components in the systems are agents. The principal architectural differences with Unity or similar systems is our layered approach to build the system as well as low-latency communication protocols and broadcast mechanisms instead of web services.

4.2 Implementation of the Distributed System

We decided to implement the system using GNUstep [94], a set of frameworks from the Free Software Foundation implementing the OpenStep specification [189], and providing a full object-oriented system, using the Objective-C language [59]. Objective-C is a high-level language implementing Smalltalk semantics [101] on top of the C language; it is, contrary to C++, a strict superset of the ANSI C.

One of the main interest of using Objective-C, in addition to its dynamic approach, and message-based object-oriented nature (something particularly suited to implement software agents), is the possibility to freely mix in the same source file Objective-C an C code (as C code is valid Objective-C), but also C++ code.

This allowed us to easily integrate pre-existing components programmed in C or C++ such as optimised renderers, simply by creating Objective-C objects containing C/C++ code calling the legacy code we wanted to use.

Using GNUstep meant that it was easy to have cross-platform compatibility and have the code running on various operating systems, such as Microsoft Windows or Apple's Mac OS X – Mac OS X's Cocoa [20] libraries being themselves an implementation of the same OpenStep specification.

The source code of our system is available on `http://www.roard.com/thesis/`.

4.3 Communication Layer

Each agent is an isolated process running on a given computer. Communications are thus an important part of our distributed system.

Figure 4.1 on the following page shows the communication architecture of an agent. Each agent contains communication centres, objects that deal with a given communication protocol. For each of these communication centres, an agent can add message handler objects that will take responsability for one particular type of message. A communication centre waits for connections, parses the type of any incoming message, and forwards the message to the corresponding message handler if there is one.

4.3.1 Default Communication Centres and Communication Ports

Each agent maintains a list of its communication centres and their connection ports. By default, every agent supports two communication centres, both using PLIST-formatted messages: one using TCP, exchanging reliable unicast messages, and one using UDP, receiving and sending multicast messages.

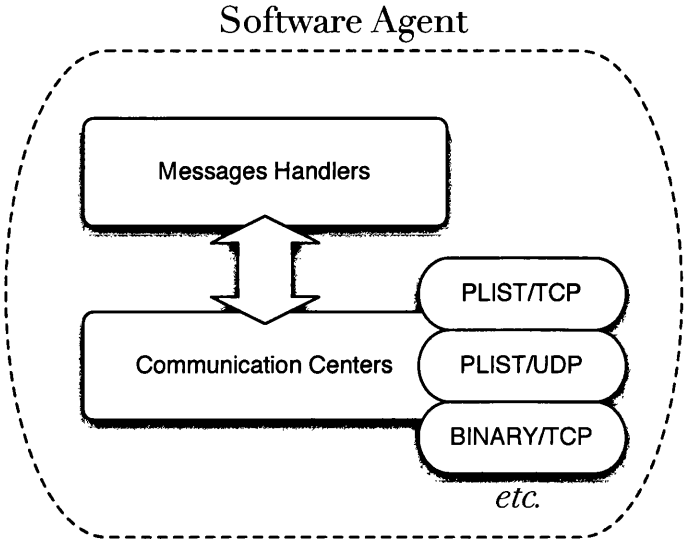


Figure 4.1: *Communication architecture of a software agent*

An important aspect of the communication system, and a main difference from the early system described in [212], is that the configuration and set up are done autonomously.

Communication ports are found at runtime by the agents, and not assignated by a central authority. The algorithm to find available ports consists in creating a server on a particular port and incrementing of the port counter to try the next available port if the creation of the server fails. In addition to that basic port discovery mechanism, we use system-specific methods to find the local IP address. Those two mechanisms permit an autonomic configuration.

The final result of this configuration process has agents listening on a specific TCP port; agents can (and will if they are running on the same machine) have different ports. Communication can still occur without any fixed port by using the discovery process described in section 4.5.1 on page 87.

4.3.2 UDP and Broadcast Communication

Connectionless communications based on UDP have numerous advantages for a visualisation system [30]. Another interesting aspect of using UDP is the possibility to easily broadcast messages (on the local network). Figure 4.2 on the following page shows such a use of broadcasting by our system, displaying a real-time maximum intensity projection (MIP) of the CT head dataset simultaneously on three different devices.

Figure 4.2: *Example of a broadcast communication: the image is generated on a cluster and visualised simultaneously on three different computers*

Having a broadcast mechanism also has other uses than just sharing a final rendering between different clients.

In fact, this mechanism is a particularly important one in our system, as it is used for discovery of agents, in a similar manner as a multicast DNS in Zeroconf [266] is used for service discovery. The actual mechanism is described in section 4.5.1 on page 87.

4.3.3 Federation of Agents

Agents usually interact with other agents on the same local network, which permit us to use a costless broadcasting (*i.e.*, without additional cost compared to sending a normal message) : being on the same physical network, we simply used UDP to send packets that any nodes could intercept.

When agents located on different local networks need to interact, a gateway agent is needed (Figure 4.3 on the following page). Gateway agents need to be visible from the other networks. When agent **A** sends a message to agent **E**, the communication layer checks **E**'s address, realizes it is a different one and thus sends the message to the gateway agent. The gateway contacts its sibling and sends the message, which is then forwarded to **E**. Note that broadcast queries can optionally be forwarded too.

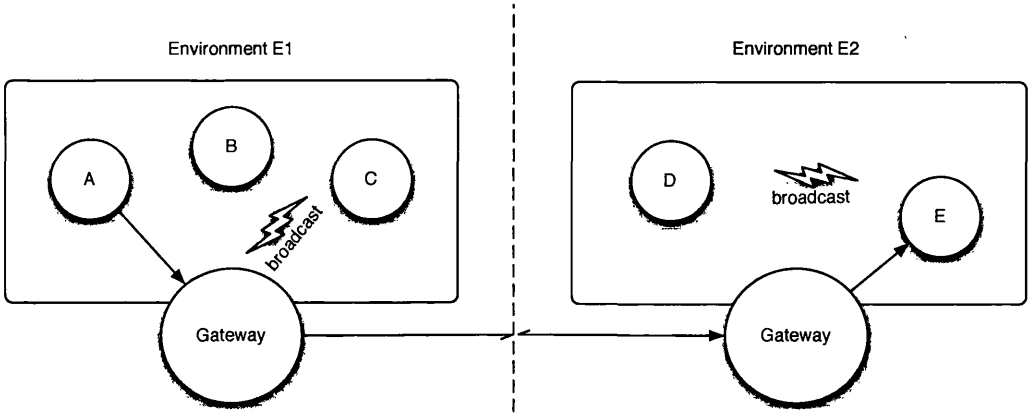


Figure 4.3: *Federation of agents*

Figure 4.2 on the previous page illustrates this mechanism; the actual image rendering is done on a cluster of machines using a private range of IPs; the cluster is only accessible through a single public node. The three machines displaying the rendered image are on a same network. A gateway agent is in place between the public node of the cluster and the local network used by the machines, and forwards image requests and image results, which are then broadcast. Also note that one machine is running FreeBSD (virtualized), one Mac OS X, and the tablet runs a variant of Linux.

4.3.4 Communication Encoding: the PLIST Format

In a multi-agent system, communication between agents is done by exchanging messages; the way messages are represented and encoded is an important design choice.

We wanted to exchange structured information easily, possibly between processes written in different programming languages. It was therefore important to keep the serialization format simple and easy to implement. A typical choice would have been to use an XML encoding, using either XML-RPC or SOAP (we did use XML-RPC in our first prototypes); we settled instead on a modified[2] property list, or PLIST [188] format.

The PLIST format is a simple way of serializing structured information in ASCII, rather similar to the JSON [61] format. The principal advantage of using an ASCII format is easy interoperability, better debugging capabilities, and in general simpler programming. In contrast to XML, this format is also simple enough to be readable

[2]the modification consists in processing the resulting ASCII representation to remove any end-of-line characters so that the final message is fully contained on a single line, which simplifies later message handling.

in raw form without any tool, and much less verbose – an important point for minimizing latency (see section 4.3.4.1).

The existence of efficient implementations in both GNUstep, Cocoa and NetBSD of this encoding format made us choose it over JSON, although integrating JSON would now be rather easy (many JSON libraries are now available).

The following shows basic types using PLIST encoding.

String

Strings are represented as:

```
"This is a string"
```

Arrays

Arrays are represented as:

```
( "Monday", "Tuesday", "Wednesday" )
```

Dictionaries

Dictionaries are represented as:

```
{
    "key1" = "value A";
    "key2" = "value B";
}
```

4.3.4.1 Comparison Between XML-RPC Encoding and PLIST

As explained above, one interesting aspect of the PLIST format is its small overhead. As an example, here is the encoding of a simple structure first using the XML-RPC encoding scheme:

```
<array>
   <data>
      <value><i4>12</i4></value>
      <value><string>Egypt</string></value>
      <value><boolean>0</boolean></value>
      <value><i4>-31</i4></value>
   </data>
</array>
```

And the same information represented with the PLIST encoding:

```
( 12, "Egypt", 0, -31 )
```

The XML-RPC encoding uses 153 characters (without spaces) for 18 characters for the PLIST encoding, *i.e.* a $\times 8.5$ improvement factor. While it is possible to reduce the number of characters of the XML-RPC encoding (using simple compression schemes such as LZH), this obviously adds some overhead. Writing a parser for PLIST is also simpler as structure qualifiers are limited to one character ({} () ; , =).

Here is a second example illustrating a full XML-RPC request:

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181

<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
      <value><i4>41</i4></value>
    </param>
  </params>
</methodCall>
```

Which uses 276 characters for this simple request. Here is the same request using our message passing encoding:

```
{
  kind="request",
  name="getStateName",
  params= { state=41; }
}
```

Amounting to 55 characters (still a $\times 5$ improvement over XML-RPC).

4.4 Scripting Integration

Having a short as possible development loop for programmers is an important asset; it allows more organic development and short programming iteration cycles. An important part of programming environments like Lisp or Smalltalk is the availability of a so-called *REPL* (Read-Eval-Print-Loop) facility. It usually takes shape as a shell that let the programmer create new code on the fly, evaluate it, etc. More importantly to us, it is also a powerful introspection capability, allowing programmers to inspect and even modify running code.

In order to provide a similar mechanism, we used StepTalk[250], a scripting framework for GNUstep and Mac OS X, to provide scripting capabilities in our system. The two principal goals we wanted to use StepTalk for were:

- To let users program complete agents using a script language rather than Objective-C or another compiled language
- To have a shell interface letting users easily query the system at runtime

StepTalk is a general framework which provides scripting capabilities to applications. It provides by default a few language plugins[3], notably a Guile and Smalltalk interpreter.

We chose to use the Smalltalk language as it maps perfectly with the Objective-C semantic[4] and provides us with a cleaner and simpler syntax (see the Smalltalk introduction chapter page 194).

Agents can be written as a text file with the executable bit set and containing the following first line (using a common unix loader feature to pass control to the shell):

```
#!/usr/local/bin/AgentInterpreter
```

AgentInterpreter then parses the file and creates an equivalent Objective-C object. Objective-C being a dynamic langage based on objects sending messages at run-time, it is possible to use the Objective-C runtime library to create a "fake" object that knows how to answer the messages it receives. In our case, AgentInterpreter automatically creates an object that will forward the messages defined by the Smalltalk code to the StepTalk interpreter to be executed. As such, the bridge between a "Smalltalk object" created via AgentInterpreter and a normal Objective-C object is

[3]It is possible to add new language plugins – an Io interpreter was made available for example – although one must take note that StepTalk principally targets dynamic languages

[4]Objective-C and Smalltalk share semantics, for obvious historical and design reasons, as Brad Cox's goal with Objective-C was to implement Smalltalk semantics on top of the C language

transparent: for any other objects in the runtime, our interpreted object is for all intended means a perfectly normal object.

When AgentInterpreter creates an instance of the object, the method *initialize* is called, giving the object the opportunity to initialize itself; the object has access to the complete agent environment, can define new message handlers, etc.

Figure 4.1 shows an example of a simple agent created that way and answering to a 'ping' message.

```
0 #!/usr/local/bin/AgentInterpreter

  !!

  initialize
5     agentName ← 'Pong'.
      self answersMessage: 'PING' with: 'ping:on:'.

  !!

10 ping: info on: socket
      socket writeLine: 'PONG'.
  ↑ true

  !!
```

Listing 4.1: Fileout example: a simple agent answering to a ping message by setting an automatic message handler

4.5 Agent Mechanisms

In addition to the basic communication features presented in the previous sections, agents need specific mechanisms such as a way to discover other agents, establish and maintain relationships with them, or easily distribute computation among groups of agents. The following sections will detail these mechanisms.

4.5.1 Agent Discovery Mechanisms

Agents can send broadcast queries that any other agents on the same broadcast channel will receive. In that way, no agents need to know about the others until it is necessary; more importantly, no central server is needed.

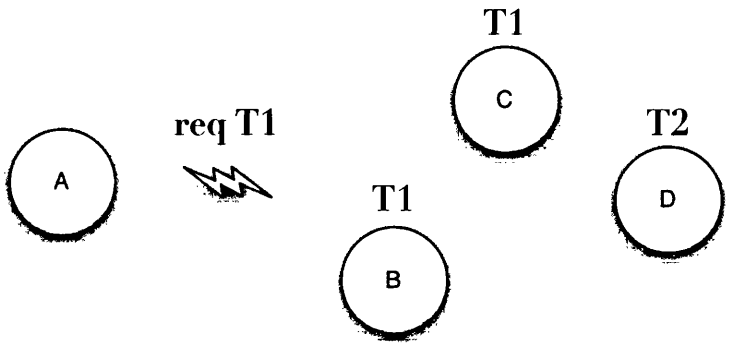


Figure 4.4: *Agent A broadcast a need for agents of type T1*

An agent **A** that needs another agent of a type **T1** sends a broadcast query on the network (fig. 4.4); upon receiving such a request, and if available and electible, agents send back a reliable PLIST TCP message to the requester containing a list of its communication centres with their associated ports and its own IP address (fig. 4.5).

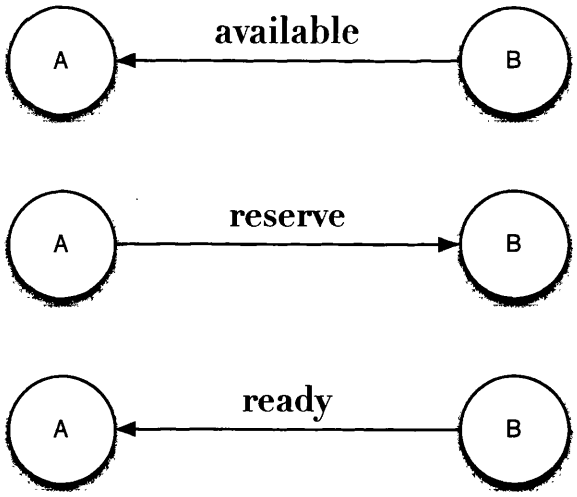


Figure 4.5: *Available-Reserve-Ready conversation*

The requester agent can then add the agent localization information to an internal list and use this information to send messages to it, possibly choosing the protocol (*e.g.*, instead of using the generic PLIST communication protocol, use a binary protocol if the receiving agent supports it).

Every agent keeps a list of the agents it communicates with and the agents it is connected to. Depending on the agent, it can accept being used by more than one agent or not, and thus signalling itself as available or not.

4.5.2 Agent Creation

When an agent sends a request for another agent, if an instance is available it signals itself to the requester. But if no instance exists in the system, a special agent takes over: a factory agent.

Every node in the cluster runs a single instance of a general factory agent. It is the only configuration needed to add a node in the agent system. This general factory agent listens to agent creation requests sent on the broadcast channel, and starts agents upon demand.

4.5.3 Agent Relationships

Each agent keeps a list of the other types of agents it needs, and a list of instances of corresponding agents. Every time a message is sent, a new connection is made – there is no keep-alive connection between messages. Not having keep-alive connection simplifies failure recovery and the writing of agents, as a message either goes through or not, and the agent can immediately act upon failure.

While agents can communicate explicitly with a specific agent, they usually communicate through a relationship abstraction; agents can specify relationships with generic types of agents they need to fulfill their task. A relationship link exposes to the system the cardinality of the relation:

- one-to-one link
- one-to-*n* link, with a given *n* instances of agents
- one-to-many link, with an undefined number of agents

Describing a relationship using the agent framework is straightforward.

for a one-to-one relationship:

```
self needsAgentOfType: 'aType' named: 'anAgent'.
```

one-to-*n*:

```
self needs: 5 agentsOfType: 'aType'
     named: 'aFixedPoolOfAgents'.
```

one-to-many:

```
self needsAgentsOfType: 'aType' named: 'aPoolOfagents'.
```

Subsequent code can then send messages to the relationships:

```
anAgent ← (self relation: 'anAgent').
anAgent sendMessage: 'PING'.
```

```
poolOfAgents ← (self relation: 'aPoolOfagents').
poolOfAgents sendMessage: 'PING'.
(poolOfAgent nextAgent) sendMessage: 'PING'.
```

Note: when using the pool of agents, the first call to `#sendMessage:` will send it to every agent in the pool. The second call is sent to the result of `(poolOfAgent nextAgent)`, thus to a single agent in the pool only.

4.5.3.1 Relationship Reliability

This higher-level abstraction lets the system be in charge of maintaining the relationship between agents. As every message is a new connection, an agent does not have to keep track of specific instances of the agents it communicates with – in fact, for two consecutive messages there is no guarantee that the receiving agent will be the same (*e.g.* the receiving agent can crash or be inaccessible and another instance be called instead).

Agents usually send information (state) to other agents during the Available-Reserve-Ready handshake (fig. 4.5 on page 88), and to update runtime state changes.

For example, if we have an agent **A** needing another agent of type **T** (*i.e.*, a one-to-one link), and two available agents **B** and **C** of type **T**; when **A** broadcasts a need for an agent of type **T**, both **B** and **C** signal themselves to **A**, but only one will be accepted (*e.g.*, **B**).

A will send a state S_i to **B** during the handshake. Let's say that the original state is S_1; if the state changes to S_2, **A** will send a message with S_2 to **B**, but will also use S_2 if a new Available-Reserve-Ready handshake is done.

If **B** becomes unavailable, a new broadcast request will be automatically sent, **C** will receive it and signal itself to **A**, complete the handshake, and be used instead of **B**, in a completely transparent manner for **A**.

4.5.3.2 One-to-one Link

A one-to-one link is the simplest case of a relationship; agents indicate the type of agent they want to communicate with. The system will ensure that the link will be available (as described in the previous section). If no agent exists, one can be created (see section 4.5.2 on the previous page).

Though problems are detected at the message level, agents can also indicate that a relationship has to be monitored constantly (*e.g.* if the usual conversation is sporadic). Detecting a fault only at the message level could be a problem if no agents

are available – a new agent would need to be restarted, which can possibly have a high starting up time. Monitoring constantly the relation can help shorten that delay.

In that case, the framework registers the relationship to a monitor agent, which regularly pings the agents and signals any problems. There is only one monitor agent per cluster node.

4.5.3.3 Pool of Agents: One-to-*n* and One-to-many Links

A one-to-*n* relationship is a limited version of the one-to-many relationship. Apart from limiting the number of agents, they work in the exact same way. The receiving agent has a pool of agents at its disposal rather than a single one.

The number of available agents is bounded by $0 < i \leq n$ in the one-to-*n* configuration, or $0 < i$ in the one-to-many configuration.

The protection mechanism, if enabled, works the same way as in the one-to-one relationship, trying to keep n instances active for the one-to-*n* relationship, and trying to keep as many instances ever connected with the one-to-many configuration.

4.5.3.4 Listeners and Pipes

An agent can register with other agents as listener for specific events;

```
agent ← self relation: 'anAgent'.
self registerForEvent: 'PingReceived' on: agent.
```

A similar mechanism is used to implement pipelining, by using the specific event *output*. Let's take an example where we have a pipeline from $A \rightarrow B \rightarrow C$, in agent A we indicate that when we call the method **#output:**, its parameter method will be sent to the agent B:

```
" we are in A "
self outputTo: B.
```

The Smalltalk parser was modified to allow for a simpler syntax; instead of writing **#output:** one can simply writes -> instead (this is expanded automatically to **#output:**).

```
A outputTo: B.
" equivalent notation : "
A ↦ B
```

4.5.4 Map/Reduce Implementation

The Map/Reduce paradigm (see 2.2.6.2 on page 30) became in recent years a popular way of architecturing parallel code, with the notable consequence of lowering the complexity of writing such code. As such, having an implementation of this paradigm (and the necessary system mechanisms) was a worthy feature to have in our system, providing programmers with a familiar paradigm.

Another advantage of making Map/Reduce a first-class feature of our system was also to provide a standard way of expressing parallel computation. In addition, this allowed us to provide built-in and transparent features around Map/Reduce such as failure recovery and automatic resource discovery (leveragin existing mechanisms implemented in our system).

Most notably, we were able to use Map/Reduce to implement our distributed rendering mechanism, adapting it to low latency and high framerates conditions (more details are provided in section 7.7 on page 143).

Our implementation of Map/Reduce works with agents implementing the Map, Split or Reduce operations; the system will automatically take care of reserving the agents and dealing with fault-tolerance (see previous sections of this chapter):

- The *Split* operation takes the original input and splits it in multiple runs (one run per remote agent doing the map operation).
- The *Map* operation will send messages (containing the runs) to the available agents in the pool; these agents will process the runs and do the corresponding computations.
- The results will then be sent back to the original agent, running the *Reduce* operation.

As an example of a Map/Reduce application in our system, we can show how to parallelise a "word count" application, where, given a list of words, we return their respective frequency (a common illustration of the Map/Reduce paradigm).

Note that for simplicity, we will consider only rather naive implementations of the different operations; more efficient methods are obviously possible.

To implement this example in our system, we need to provide the split, map and reduce operations (here implemented in Smalltalk, see figures 4.2 on the following page, 4.3 on page 94 and 4.4 on page 95).

The original input (fig. 4.6 on the next page) is first split into multiple runs using the algorithm 4.2 on the following page. Figure 4.7 on the next page shows the split input on 3 nodes N_0, N_1, N_2.

Each node can then run the map operation (algorithm 4.3 on page 94), which basically creates for each word a pair with the word and an occurence number set to

1. Figures 4.5.4 on the next page, 4.8 on the following page and 4.9 on the next page show the result of this operation.

Finally, each mapping is sent back to the agent and reduced one at a time (Figures 4.5.4 on page 95, 4.11 on page 95 and 4.12 on page 95) using the algorithm 4.4 on page 95.

Maxime Maxime Chloe Maxime Alexandre Alexandre Alexandre Alexandre Alexandre Ines Chloe Mathis Mathis Marie Chloe Chloe Nathan Nathan Nathan Clara Clara Clara Clara Clara Antoine Antoine Antoine Antoine Antoine Antoine Antoine Sarah Sarah Sarah Sarah

Figure 4.6: *Original input*

```
"split: content on: nodes"

nbNodes ← nodes count.
perRun ← (content count) / nbNodes.
perRun ← perRun intValue.

currentNode ← 0.

content do: [:name|
    node ← nodes objectAtIndex: currentNode.
    node addObject: name.
    currentNode ← (currentNode + 1) modulo: nbNodes.
].

↑ nodes
```

Listing 4.2: Split algorithm

(
N_0: (Maxime, Maxime, Alexandre, Ines, Mathis, Chloe, Nathan, Clara, Antoine, Antoine, Antoine, Sarah),
N_1: (Maxime, Alexandre, Alexandre, Chloe, Marie, Nathan, Clara, Clara, Antoine, Antoine, Sarah, Sarah),
N_2: (Chloe, Alexandre, Alexandre, Mathis, Chloe, Nathan, Clara, Clara, Antoine, Antoine, Sarah)
)

Figure 4.7: *Split input*

```
"map: names"

list ← Array new.

names run do: [:name|
    pair ← Array new.
    pair addObject: name.
    pair addObject: 1.
    list addObject: pair.
].

↑ list
```

Listing 4.3: Map example

((Maxime, 1), (Maxime, 1), (Alexandre, 1), (Ines, 1), (Mathis, 1), (Chloe, 1), (Nathan, 1), (Clara, 1), (Antoine, 1), (Antoine, 1), (Antoine, 1), (Sarah, 1))

Figure 4.8: *Mapped input (run 0)*

((Maxime, 1), (Alexandre, 1), (Alexandre, 1), (Chloe, 1), (Marie, 1), (Nathan, 1), (Clara, 1), (Clara, 1), (Antoine, 1), (Antoine, 1), (Sarah, 1), (Sarah, 1))

Figure 4.9: *Mapped input (run 1)*

((Chloe, 1), (Alexandre, 1), (Alexandre, 1), (Mathis, 1), (Chloe, 1), (Nathan, 1), (Clara, 1), (Clara, 1), (Antoine, 1), (Antoine, 1), (Sarah, 1))

Figure 4.10: *Mapped input (run 2)*

```
"reduce: map to: list"

map do: [:pair|
    name ← pair objectAtIndex: 0.
    number ← pair objectAtIndex: 1.
    currentNumber ← list objectForKey: name.
    (currentNumber isNil) ifTrue: [
        list setObject: number forKey: name.
    ] ifFalse: [
        list setObject: currentNumber + number
                forKey: name.
    ]
].

↑ list
```

Listing 4.4: Reduce example

Alexandre = 1; Antoine = 3; Chloe = 1; Clara = 1; Ines = 1; Mathis = 1; Maxime = 2; Nathan = 1; Sarah = 1;

Figure 4.11: *Reduced input (run 0)*

Alexandre = 3; Antoine = 5; Chloe = 2; Clara = 3; Ines = 1; Marie = 1; Mathis = 1; Maxime = 3; Nathan = 2; Sarah = 3;

Figure 4.12: *Reduced input (run 0 + run 1)*

Alexandre = 5; Antoine = 7; Chloe = 4; Clara = 5; Ines = 1; Marie = 1; Mathis = 2; Maxime = 3; Nathan = 3; Sarah = 4;

Figure 4.13: *Reduced input (run 0 + run 1 + run 2)*

4.6 Conclusion

In this chapter, we presented our implementation of a multi-agent system. Agents are implemented using an Objective-C framework allowing an easy integration of pre-existing C/C++ code. In addition, agents can be programmed partially or fully as a collection of Smalltalk scripts, allowing a rapid development cycle if needed.

While the general approach of using an agent system to implement autonomic features is comparable to other projects (most notably Unity [244]), one of the main difference with existing systems such as Unity is that we chose to implement our own communication methods instead of relying on pre-existing protocols such as OGSA [92] — while we lose potential ease of integration with existing systems, this was a necessity for performance reasons. Nonetheless, we kept our communication protocol open and easily implementable.

More importantly, our communication mechanisms are not just efficient implementation, they also differ in some key aspects such as the possibility to state relationships, enabling automatic failure recovery mechanisms, as well as putting broadcast capabilities at the very center of our system. Using those capabilities, the system works without any central authority, discovering available nodes and agents. New agents can be started upon demand and autoconfigured by the system instead of having to pre-configure and set up everything beforehand.

A final difference with other agent-based autonomic projects is our implementation of the Map/Reduce paradigm, which simplifies parallel computation and collaboration among agents by providing a known and easy to work with framework.

(CHAPTER ...5)

A Distributed Graphic Pipeline

"Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."
— Lewis Carroll (Alice's Adventures in Wonderland)

Contents

WE PRESENTED in the previous chapter a generic multi-agent system. In this chapter, we are going to describe how a distributed graphic pipeline can be created on top of it, taking advantage of the functionalities of the agent layer. Distributed graphic pipelines usually are implemented following a dataflow model [248, 75, 43].

Our system follows a similar pattern, with the difference that we do not use basic software components but take advantage of the multi-agent system underneath.

5.1 Architecture Overview

A graphic pipeline is a common paradigm in graphic programming, first proposed as a conceptual model by Haber and Mc Nabb [114], originating from the work of Upson et al. on AVS, the Application Visualisation System [248].

A graphic pipeline represents the visualisation process as a directed graph; nodes being processing elements, arcs representing data flow, or as Duke *et al.* remark it in [74], representing data dependencies.

Our pipeline organisation deals with a different kind of data flows, such as:

- Pipeline environment
- Image data
- Datasets

Nodes of the pipeline can intercept and process those data flows. What we call the pipeline environment groups all the state used by the pipeline to render an image: camera position, rendering quality, etc. Image data consists of raw uncompressed bytes, arranged in 2D arrays of pixels. Datasets are typed byte arrays: for example, volume datasets can be stored as unsigned integer values, or 32-bit float values.

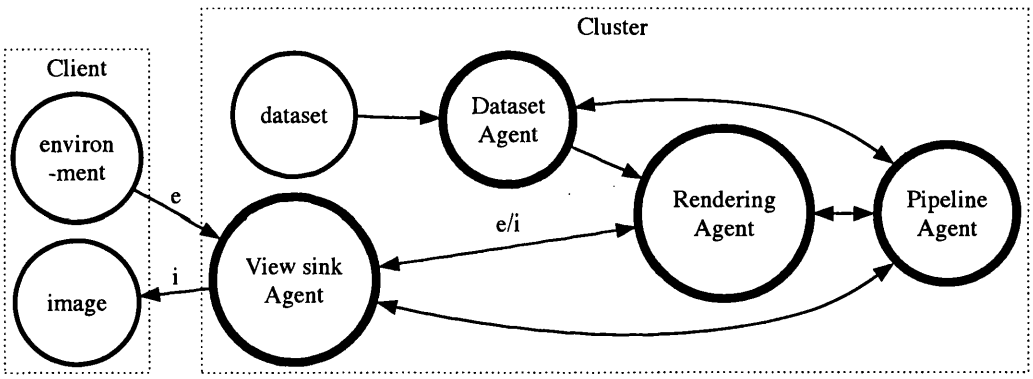


Figure 5.1: *Distributed graphic pipeline*

Fig. 5.1 shows the general organisation of the pipeline. The client part does not need to run on the same network as the rest of the system; the single connection point is the view sink agent. Commands (*e.g.* camera viewpoint) are sent to the view sink, which forwards them to the rest of the pipeline. In the simpler case (as shown in the diagram), we have a view sink agent, a Rendering agent, a Dataset agent, and a Pipeline agent. Responsibilities are as follows:

- The view sink agent is the point of entry – the gateway – it receives requests from the outside world and can return an image
- The Dataset agent is in charge of the data used by the pipeline. If necessary, it will transfer the data.
- The Rendering agent uses the data provided by the Dataset agent and the environment forwarded by the view sink agent to render an image. The image is sent back to the view sink which can then serve it to clients

- The Pipeline agent is in charge of the whole pipeline – it contains the pipeline description, along with information, added by agents, to be shared.

5.2 Agents

The following sections will detail each of the main agents composing the graphic pipeline, as well as the different data flows used.

5.2.1 view sink Agent

The view sink serves as the main interface between the pipeline and the rest of the world. A pipeline can be run on a dedicated cluster, with the only public interface being the view sink.

As an agent, the view sink is rather basic in its functionalities, acting more like a gateway to the system; it receives requests sent by clients, and forwards those requests to the rest of the pipeline, possibly after transformation or filtering. A basic feature of the view sink is for instance to throttle the image requests by only allowing the request to go through if the pipeline is ready, and only allowing requests different from the one corresponding to the last rendered image.

The other important feature of the view sink agent is to serve the image rendered by the pipeline back to the clients [1]. The view sink caches the last successfully rendered image (as well as the associated environment) and can serve it if the request corresponds to the same environment; the image being sent back is first encoded in JPEG to optimise bandwidth use. The view sink can also possibly modify the degree of compression, trading off image quality for bandwidth usage.

5.2.2 Dataset Agent

The dataset agent is in charge of managing the dataset. It keeps information about it that other agents can query, such as the data location, or more intrinsic information such as (taking a volumetric dataset as an example) its number of voxels in x,y,z dimensions, the actual format used by the dataset (*e.g.*, short int, float), density range, the voxel ratio, a thumbnail representation of the dataset, etc.

The dataset agent is also in charge of providing the data to agents of the pipeline if necessary, by organising its transfert to the agents local machines using the system copy facilities.

[1] more than one client can be connected to the view sink agent at any time

5.2.3 Rendering Agent

The Rendering agent is at the core of the pipeline; its role is to render a 2D representation of the data provided by the dataset agent, according to the values of the current rendering environment. Different rendering algorithms can be provided by different rendering agents; in our implementation, we had three agents, all working with a volumetric dataset:

- a Direct Surface Rendering (DSR) agent; given a specific density value, an isosurface is computed from the dataset and used to render a representation of the data (figure 5.5 on page 107)
- an octree rendering agent, building an octree datastructure from the dataset, allowing to change the specified isosurface in realtime
- a Maximup Intensity Projection (MIP) agent, which simply constructs an image by firing rays directly into the dataset, computing the integral and displaying the result (see figure 4.2 on page 82)

Each one of those agents provides different trade-offs for the visualisation. Notably, while the rendered result is very similar between the DSR agent and the octree, the DSR agent is slightly faster, and can accommodate much bigger datasets as it needs less memory. The octree renderer on the other hand allows for much better exploration of the dataset as the isosurface examined can be varied in real-time, while the DSR needs a preprocessing step and to reload the data.

5.2.4 Pipeline Agent

The Pipeline agent has two principal goals. First, for the rest of the agent system, it permits to identify a graphic pipeline, that is, the various agents involved in a pipeline. Basic management actions can be done on a pipeline, such as starting, pausing or stopping a pipeline.

Its second role is to keep track of the agents involved in a pipeline; those agents can then in turn query the pipeline agent to ask for another agent of the pipeline.

5.2.5 Dataflow Overview

Looking at figure 5.1 on page 98, the pipeline is composed of the various agents discussed previously; the other important aspects of a pipeline are the different data flows interacting.

The first data flow goes from the *dataset* agent to the *rendering* agent. The rendering agent asks the pipeline to point it to the relevant *dataset* agent; it then loads the

dataset managed by the dataset agent, using the information provided by the dataset agent.

The second data flow we can identify on the diagram is the environment data flow, from the client controls to the *view sink* agent, to the *rendering* agent. The environment data flow can be understood as the stream of image requests; an environment contains information such as the camera viewpoint, on which the final rendering depends. The *view sink* agent filters the request to only let through genuine new requests, in order to reduce unnecessary strain on the pipeline, so that only different environments are passed to the *rendering* agent, which then renders a corresponding image.

The third identifiable data flow is the rendered image, from the *rendering* agent to the *view sink*. The view sink transforms the raw 2D image data, converting it into a compressed JPEG image before sending it back to the viewer on the client side.

5.3 Visualisation Clients

We call clients the softwares connected to the distributed pipeline via the view sink agent (Fig. 5.1 on page 98) and allowing users to interact with the pipeline and visualise the result of the pipeline (*i.e.* the rendered images); it can be argued that the usefulness of a graphic pipeline (or any computation for that matter) is only limited by how we can use it externally.

One of our goals was to allow for various types of client to interact with the pipelines; keeping the communication protocol simple was key to that aim.

5.3.1 Rendering Loop

We can understand the general interaction between a client and the pipeline as a continous loop, with the client sending a request for an image to the pipeline, and receiving in return an image. Figure 5.2 on the next page shows this mechanism, which we call a *synchronous loop*.

As the client waits to receive the image before initiating a new request, there cannot be any type of request pile-up causing (at best) a lag and (at worse) a freeze of the pipeline as it keeps trying to fulfill requests that come faster than it can process.

This architecture has the additional benefit that the user will always see a consistent picture corresponding to their parameters [265].

It must be noted though that if the client keeps a stack of requests, rather than preventing (or discarding) new requests from the UI until the image is received, the lag effect can still appear.

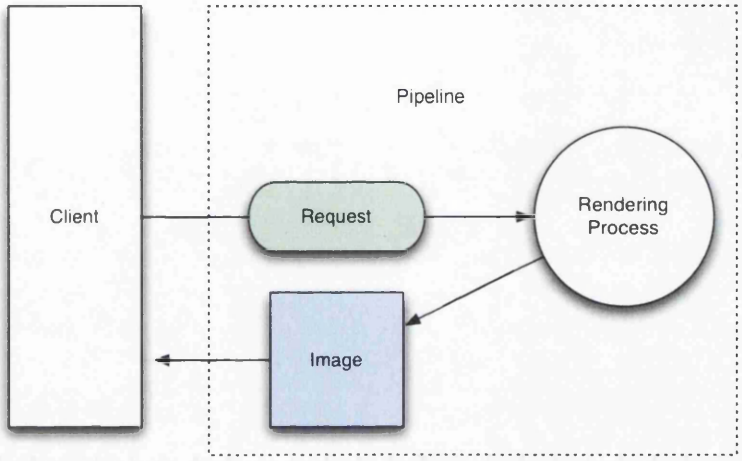


Figure 5.2: *Synchronous Rendering Loop*

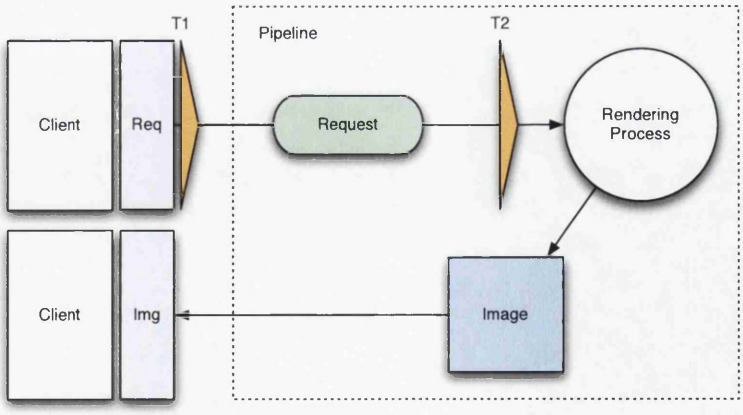


Figure 5.3: *Asynchronous Rendering Loop*

While appealing, this type of rendering loop does not address our concerns; notably, multiple clients cannot easily interact with the same pipeline without affecting their performances. The architecture we chose to implement is similar to the one shown on figure 5.3 on the preceding page. Here, image requests and images received are disconnected; both actions can be run by entirely different clients. One benefit is to be able to easily add viewer clients to a pipeline, so we can have one master client controlling the parameters and a multitude of passive viewers. This allows for example to create display walls easily.

In order to prevent any kind of pile-up, two throttling stages T1 and T2 exist. T1 is run client-side, and throttles the requests to an arbitrary number per second; this allows the client to control the visualisation in a smooth manner — imagine doing a zoom via a slider widget, if every tick of the move equals a request; the pipeline could easily get saturated.

This first stage alone does not prevent any request pile-up, as T1 could be set to a higher number than the rendering process could deal with, or simply multiple control clients could send simultaneous requests. The throttling stage T2 serves to regularly coalesce the received requests so that the rendering process avoids this potential saturation.

The rendering process, upon receiving a request, will generate an image, and immediately after will check if another request is there, and if so repeat the process. At any time, an image will thus be available to the visualisation clients: when a viewer connects and asks for an image, it thus receives the last generated image.

In contrast to the synchronous rendering loop, this means that the received image can be out of date and not reflecting the parameters of the client; in our tests this was not a real problem, as the actual behaviour is consistent with the user's expectation (*i.e.* coalescing the requests avoids the lag effect).

5.3.2 Command Protocol: Interacting with the Pipeline State

The command protocol allows agents to set the state of a pipeline as well as consulting it – *e.g.* adding or consulting information. Conceptually, it consists of transmitting key-value pairs of specific parameters used for the pipeline.

The communication consists of a synchronous Request-Result conversation:

1. the client sends the message containing a request and waits for the answer
2. the view sink agent answers back and closes the communication

5.3.2.1 Read State Message

A message can be sent by a client by specifying a dictionary encoded as a PLIST, with the key **kind** being set as **ReadState**:

```
{ kind = ReadState; }
```

When no other keys are present in the message, as in the above example, the pipeline returns the complete list of key-values representing its state. If other keys are present, only the values of those keys are sent back to the client; the view sink agent will send back a similarly encoded answer, with their latest values. The following example will receive an answer only containing the values for the keys angle and zoom:

```
{ kind = ReadState; angle = 220; zoom = 50; }
```

5.3.2.2 Set State Message

A Set State message is similar to a Read State message, with two differences:

1. the key used is **SetState** instead of **ReadState**;
2. the values contained in the message will be used by the pipeline and possibly trigger the generation of a new image (if they are different to the last values used)

5.3.3 Accessing the Images

The protocol used to access the last image rendered by the pipeline is as simple as possible, as we wanted the possibility to facilitate the realisation of visualisation clients.

A client only needs to initiate a TCP connection on a view sink agent image port and waits for an answer. The answer is twofold:

- a 4 byte header containing the size of the returned image as an unsigned int value
- the image data, encoded in JPEG

The simplicity of this protocol allowed us to integrate new clients into the system. The image returned is always the last rendered image.

We chose to use the JPEG format as the default format to return images, as it is ubiquitous, and offers good compression ratios (and we can easily tune those ratios to impact the image size).

Other image formats can be used as easily, in particular the PNG format allows good compression ratios, without using a lossy compression (contrary to JPEG), something possibly mandatory depending on the use case. Our current pipeline implementation can switch between both formats, for the Java and the OpenStep clients.

Combining Setting a State with Accessing the Image

It is sometimes useful (to reduce the round-trip times, *i.e.* the latency) to set a state triggering a new image and wait for the image synchronously. The protocol used in that case simply combines the **SetState** command with the image access command.

The conversation is as follows:

1. the client connects to the view sink agent
2. the client sends a **SetAndRead** state message on a single line, formatted as a PLIST (similar to the **SetState** message)
3. the client waits for the next 4 bytes indicating the image size
4. the pipeline generates a new image
5. the pipeline sends the image size followed by the image data
6. the client reads the size then the image
7. the client closes the connection (optional)

While the cycle appears synchronous, it is still asynchronous; the coalescing mechanism can mean the client will update the last request on the pipeline side, and will receive the last generated image, which may not be the one corresponding to the request.

5.3.4 Visualisation Clients

We wrote different visualisation clients, connecting to the pipeline and displaying the current image. The following subsections will present some of the implementations of such clients.

5.3.4.1 A PDA Client

One of the first implementations used the Squeak platform, a free implementation of the Smalltalk Virtual Machine [136].

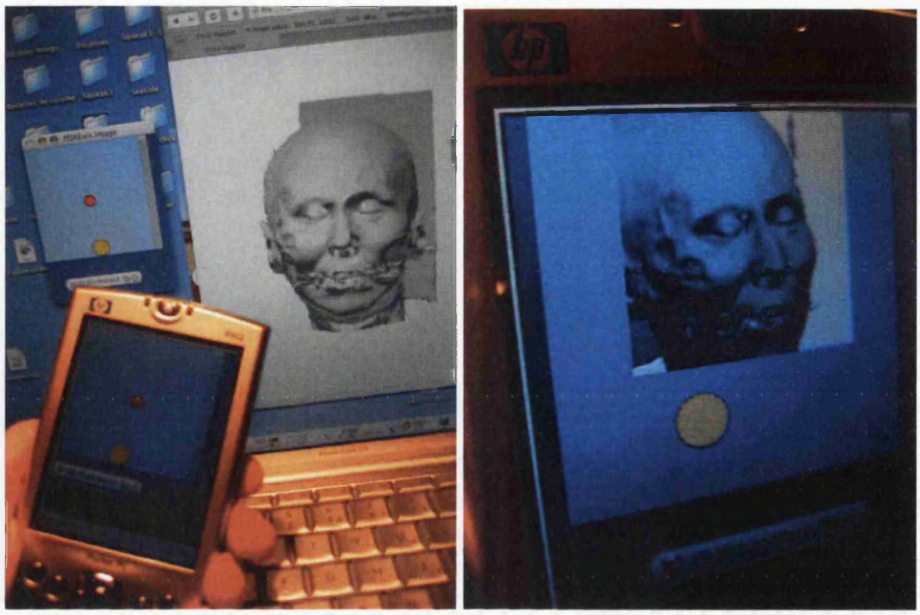


Figure 5.4: *The* PDA *and Web clients, with the* PDA *acting as a remote control on the left, and as a visualisation and control client on the right*

The flexibility of the VM allowed us to use it for writing various prototypes of the graphic pipeline. One notable feature of Squeak is its cross-platform capabilities; we used an existing port of the Squeak VM on Windows CE to run a visualisation client on a PDA.

Figure 5.4 shows that clients can choose to implement subsets of the client commands; here the PDA can be used as a simple remote control (without a visualisation surface), or be used to visualise datasets on its own.

5.3.4.2 OpenStep Client

We wrote a visualisation client using the OpenStep framework, allowing us to run it on Mac OS X (via the Cocoa implementation), Linux and Windows (via the GNUstep implementation).

Figure 5.5 on the following page shows how the client looks, using a 512×512 display. In addition to connect to a specific pipeline[2], this client has graphing capabilities to allow users to measure the run-time rendering performances (the *Perfs* button) and agent allocation (the *Agents* button).

[2]Here the client directly connects to a specific instance of a pipeline using a ssh tunnel, hence the local IP address displayed

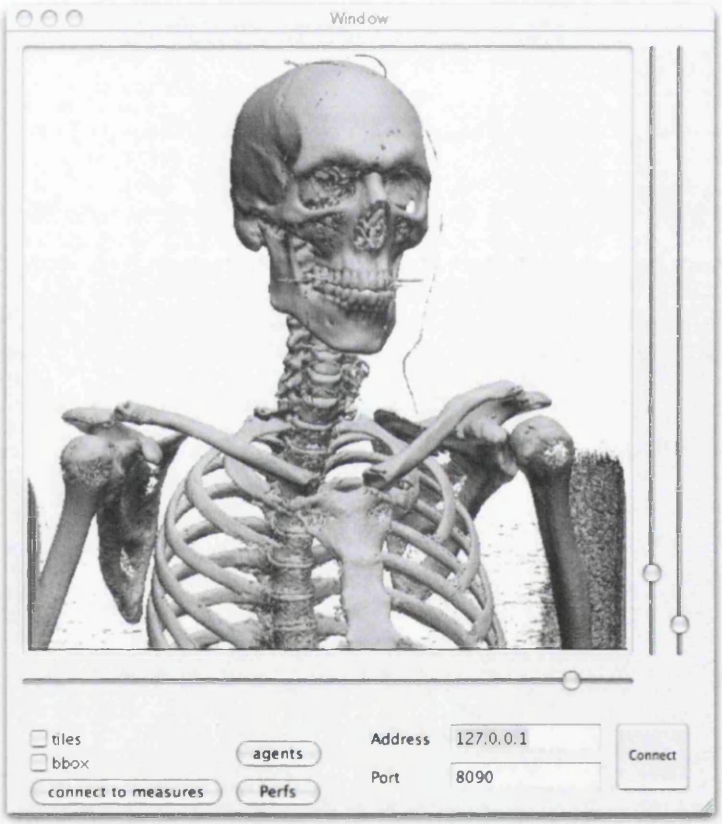


Figure 5.5: *The OpenStep client (here running on Mac OS X, visualising the Visible Human dataset)*

5.3.4.3 Java Applet Client

This client is a raw visualisation client implemented as a Java applet. The principal reason for this client is to permit embedding the client in a web page (figure 5.4 on page 106 on the left and figure 5.6), which allows us to have much richer possibilities in the way we organise the user interface, as well as lowering the entry barrier and generally allowing more flexibility [38].

5.4 Web Interface: Towards a Reflective User Interface

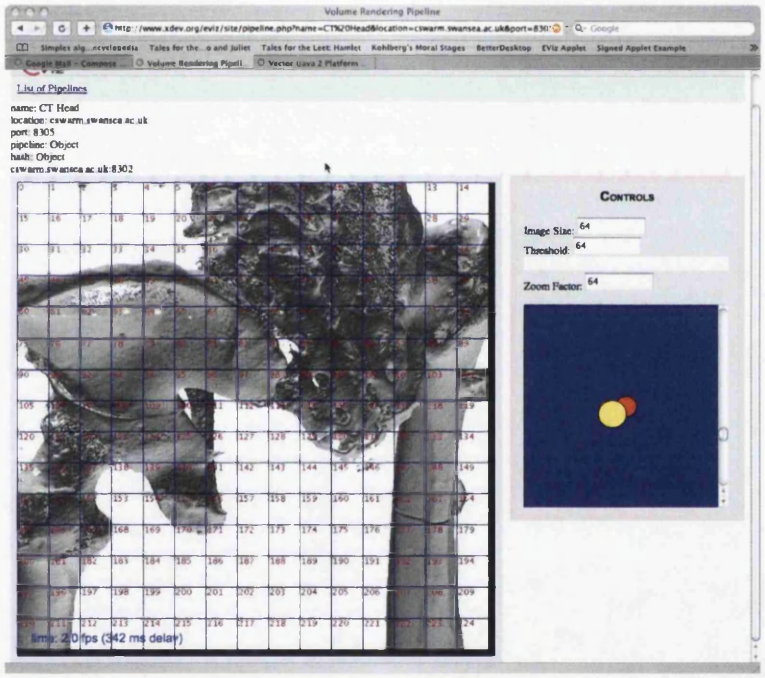


Figure 5.6: *The Web user interface*

Figure 5.6 shows the last iteration of the web user interface used to manipulate and connect to pipelines, as well as viewing datasets.

An interesting aspect of this web interface is that it is generated on the fly by a special agent, acting as a web server (implementing the HTTP/1.0 protocol), and communicating with the other agents in the system to create the web page.

We took advantage of the communication architecture of our software agents (see figure 4.1 on page 81) to add a new communication centre handling messages used to create the web page, with agents sending back blocks of HTML.

The 'web server' agent only serves as an aggregator – the content of the web pages is in fact generated by various other agents. For instance, the default web page

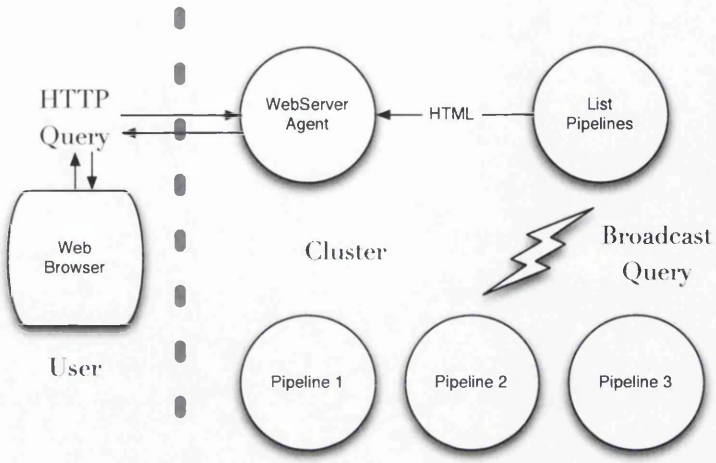


Figure 5.7: *Generating a web page listing the available pipelines in the system; all circles represent agents*

shows a list of pipelines available on the system. The 'web server' agent by default presents a simple HTML page containing only one agent 'listPipelines' (figure 5.7). The HTML page shown on figure 5.6 on the preceding page is generated by asking the pipeline agent for the corresponding web content.

An interesting possibility of this architecture is to directly return the last image rendered by the pipeline; this would simplify the user's selection of a pipeline, by showing this image along the pipeline information. This capability could also be used to create a poor man visualisation client (using auto-update javascript, or using multi-part JPEF where the network connection is kept open and new images are continuously pushed).

5.5 Performances Evaluation

We described in the previous sections the general mechanisms used to implement a graphic pipeline on top of an agent system, as well as different visualisation clients used with the system. The architecture of the graphic pipeline is open and easily extensible; however, the most open and flexible system would be useless in practical terms if the performances were not good enough.

Section 3.3 on page 62 described some common metrics used in evaluating parallel systems. We will use those metrics here to discuss the performances of the graphic pipeline.

5.5.1 Pipeline Architecture: Distributing the Rendering Load

The pipeline architecture is more complex in real life than the diagram 5.1 on page 98; notably, the rendering agent ususally corresponds to an assemblage of many different agents, which we will cover in more detail in the next chapters.

In terms of performance though, it is interesting to detail the overall organisation to understand the system's behaviour.

The aim of the pipeline is to allow visualisation of data. In our case, we worked with volumetric datasets, which by definition tend to be large and complex to render. The pipeline role is thus also to distribute the rendering load across multiple nodes in the cluster.

We chose to implement an image-based distributed rendering to visualise those datasets; multiple rendering agents take ownership of a part of the total image to be rendered. Those rendering agents are executed on different nodes on the cluster, thereby distributing the total rendering load across the cluster nodes.

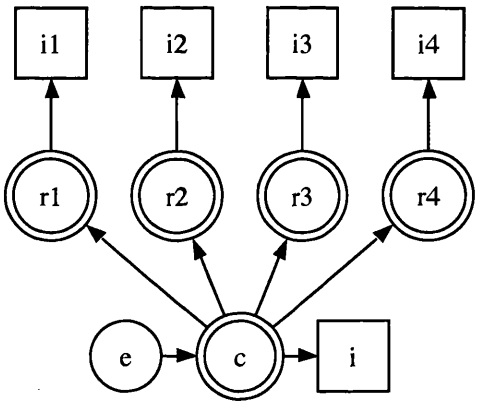


Figure 5.8: *Single-layer pipeline architecture running on a cluster of machines*

Figure 5.8 shows the general architecture of a simple distributed rendering pipeline. The pipeline transforms a request e (containing information about the rendering) into an image i, through the processing done by an agent, here, c.

This agent c is in fact a compositing agent, whose task is not to render the image directly, but to issue requests to the rendering agents — here, r_1, r_2, r_3 and r_4.

The compositing agent then gathers partial images (i_1, i_2, i_3, i_4) from those agents, and recomposes the final image i with them.

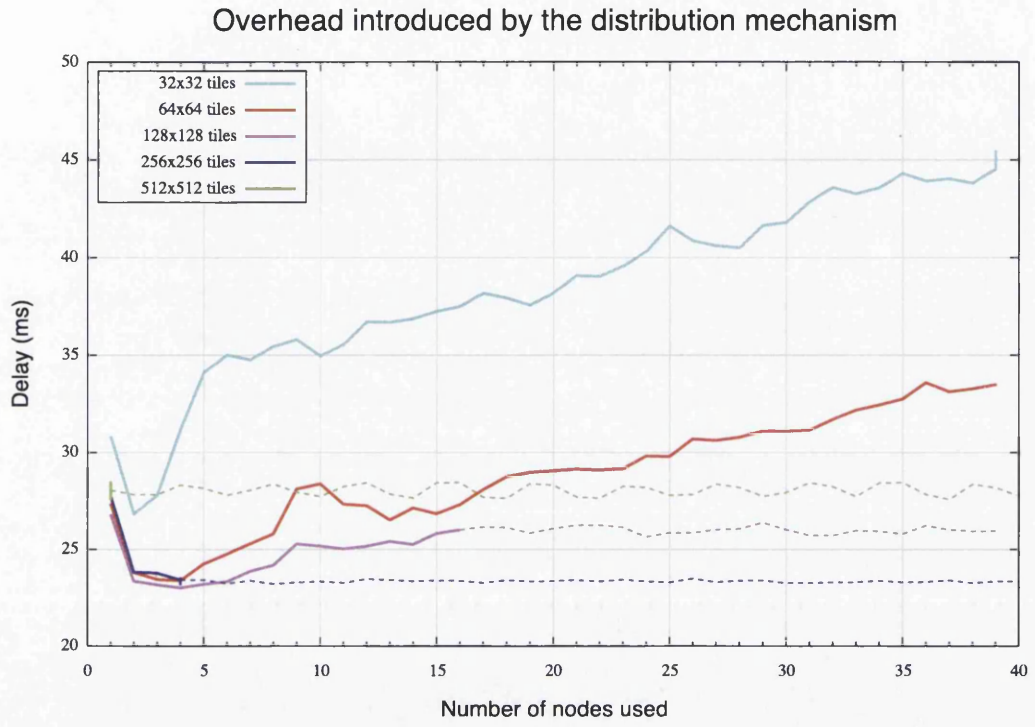


Figure 5.9: *Distribution overhead on a parallel rendering of a* 512×512 *image*

5.5.2 Experimention: measuring the system overhead

To measure the performances of the system, we implemented a "blank" rendering agent, which simply returns a blank image of the requested size. By using this rendering agent instead of a real one in our visualisation pipeline, we can measure the speed taken by the pipeline to generate an image when the rendering time is null; the measured time takes in account the entire mechanism (distribution, recomposition), minus a rendering time. This gives us the minimum amount of time taken by the pipeline to generate an image given a number of nodes, a final image size and the tiles size used.

We consider this minimum amount of time to be the inherent overhead of the system.

Figure 5.9 shows the results measured on distributing the rendering of a 512×512 image, using several size of tiles (32×32 pixels, 64×64, 128×128, 256×256, 512×512). The measures were repeated a hundred times and averaged for each combination of number of nodes and tile size. The dashed lines on the graph simply show the performances of the system when the number of available nodes depassed the number of tiles needed by the pipeline (*e.g.* using a 128×128 size of tiles equates for a maximum of 16 tiles for the 512×512 image size we used).

We can see on the figure that for a reasonable tile size of 32×32 on a 512×512 image,

the minimum delay introduced by the system is around 45 ms for 40 nodes, *i.e.* a maximum performance of around 22 fps. For the same total image size, using 64×64 tiles only introduces an overhead of 33.5 ms, *i.e.* a maximum performance of 30 fps. The reason for this additional overhead is the increased number of transmissions; with 32×32 tiles, we have a total of 256 tiles to transfer and compose; for 64×64 tiles, we only have 64 tiles to process.

5.5.3 Performance Analysis

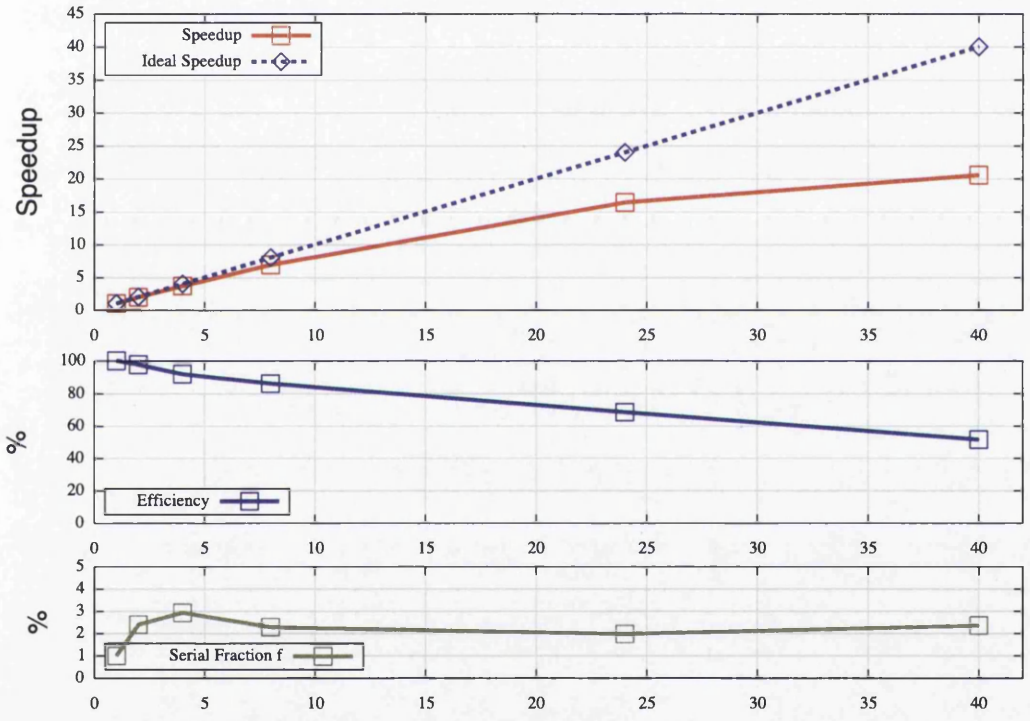


Figure 5.10: *Performances of a parallel rendering of the Visible Human dataset, using a* 512×512 *image divided into* 32×32 *tiles. The top graph shows the rendering speedup along with the ideal speedup (dashed line). The middle graph shows the progression of the parallelisation efficiency with the number of nodes. The bottom graph shows the level of the serial fraction, which is staying in the 2-3% range.*

The previous section discussed the inherent overhead of the system. But how does the system behave when running a real distributed graphic pipeline?

We can analyse real performances using measures gathered while rendering a 512×512 image of the Visible Human dataset, a 600 MB volumetric dataset. For each value, we rendered the final image a thousand times using 32×32 pixels tiles.

Those measures as well as computed metrics we discussed in Chapter 3 — speedup,

efficiency and serial fraction — are shown in Table 5.1. The metrics will let us analyse more finely the behaviour of the system.

Nodes	Time (μs)	Images/s	Speedup	Efficiency (%)	Serial fraction (%)
1	1042752	0.959	1.0	100	-
2	533902	1.873	1.953	97.6	2.400
4	283768	3.524	3.674	91.8	2.933
8	151423	6.604	6.886	86.0	2.285
24	63572	15.730	16.402	68.3	1.981
40	50697	19.725	20.568	51.4	2.358

Table 5.1: *Performance analysis*

Figure 5.10 on the preceding page shows three graphs illustrating the speedup (top graph), the efficiency (middle graph) and the serial fraction (bottom graph) we computed. We can see that while the efficiency of the system decreases regularly with the number of nodes, the serial fraction stays mostly flat, as it should; this is an indication that the system does scale properly, but is limited by the serial fraction.

5.5.4 Two-layer Pipeline Architecture

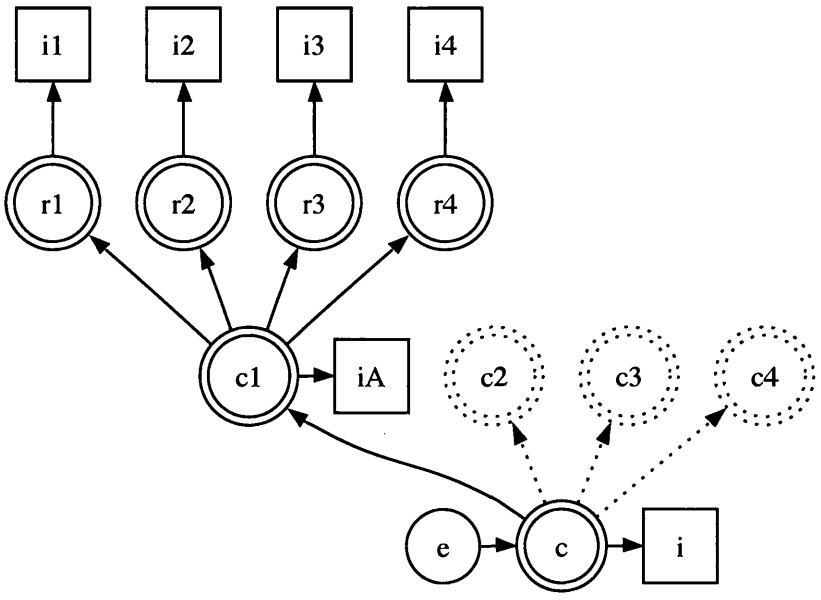


Figure 5.11: *Two-levels pipeline architecture running on a cluster of machines*

Figure 5.9 on page 111 shows the inherent overhead of the system increase when using more tiles, as the additional communication links slow the system. The main

reason is that the communication is centralised on the compositing agent *c*, as shown on figure 5.8 on page 110.

Image composition is a known bottleneck in distributed rendering systems [191], as we highlighted in 2.3.3.1 on page 39.

We thus addressed the composition problem by leveraging our software architecture, introducing a second layer of compositing agents alleviating the communication burden per node.

Figure 5.11 on the preceding page shows the architecture of this two-layer approach, with the compositing agent *c* communicating with *other* compositing agents, *c1*, *c2*, *c3* and *c4*. While the diagram only details the architecture for *c1* to keep things simple, the other intermediate compositing agents obviously mirror *c1*'s architecture.

The intermediate compositing agent *c1* keeps a pool of rendering agents available (*r1*, *r2*, *r3*, *r4*), each of them rendering a piece of the final image (*i1*, *i2*, *i3*, *i4*). Those intermediate images are gathered by *c1* to create another intermediate image *iA*, which is then sent back to *c*, which then generates the final image *i*.

This two-level architecture is necessary as it allows distributing on different nodes of the cluster not only the rendering agents, but also the compositing agents; this allows us in turn to better distribute the load involved with the image composition, and the communication with the rendering agents; it is indeed faster to transfer and process large tiles (partial images) than transmitting many small tiles.

Figure 5.12 on the next page compares the two styles of systems, using a 512×512 image divided into 1024 tiles. As can be seen, the dual layer system performs much better in terms of overhead, speedup and efficiency.

Table 5.2 on page 116 shows the values used. The overhead measured is the system overhead as defined in the previous sections, that is, the time necessary to perform the distribution of the tiles, the generation of the partial images (without any rendering, we use a "blank" image rendering agent), and the final image recomposition. The normalised time is calculated using the following formula, using the timing difference ponderated by the number of processors used:

$$t(p) = \frac{1000 + (o(p) \times p - o(1))}{p}$$

With o the measured overhead in milliseconds and p the number of processors used. This allows us to analyse how the system behaves in terms of speedup, efficiency and serial fraction.

The impact on the speedup can be seen as a consequence of the reduction of the serial fraction with a dual layer system; since the parallel fraction of the system is more important than with the single layer system, the system scales better. Note

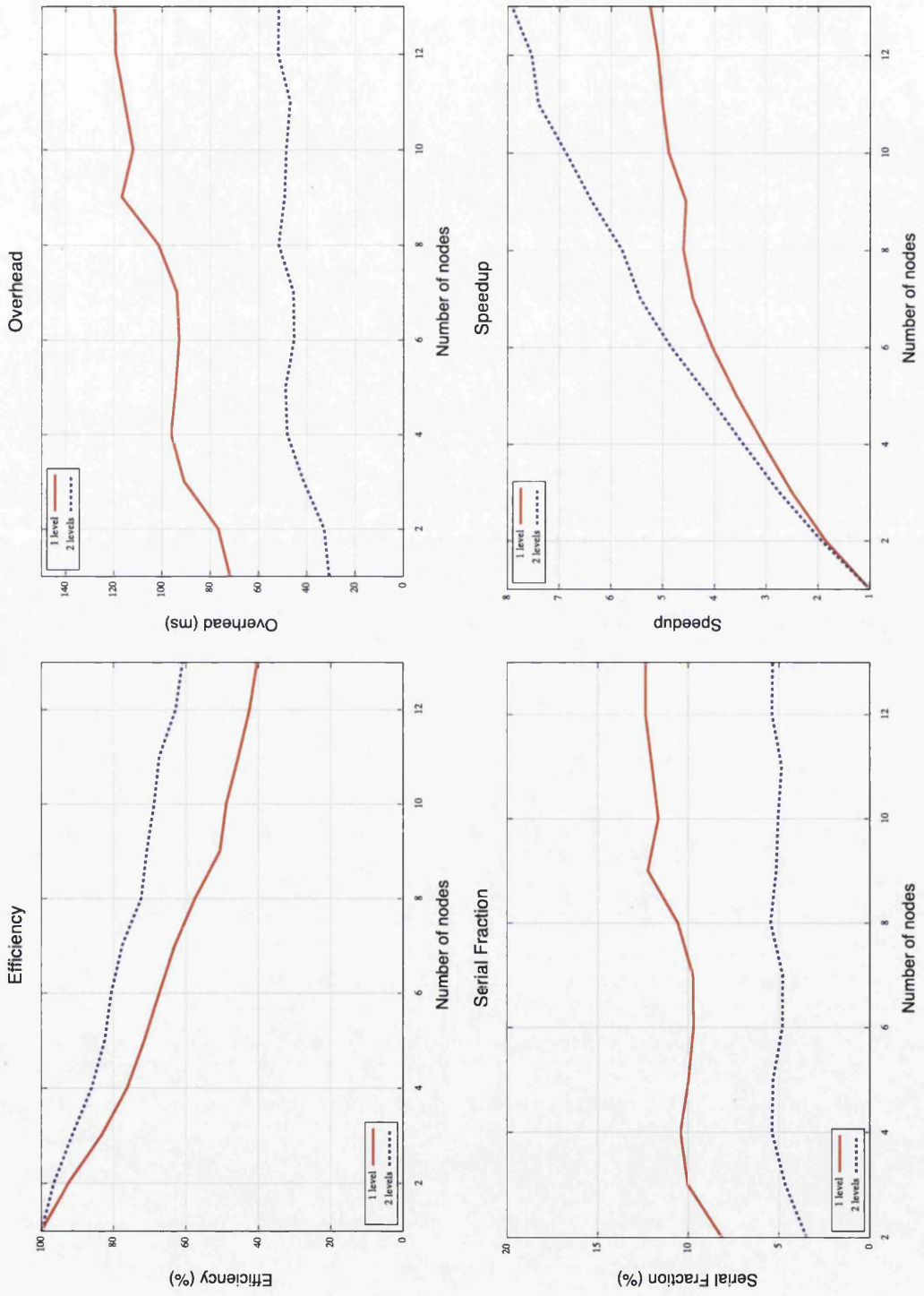


Figure 5.12: *The above graphs shows the overhead, speedup, efficiency and the serial fraction when using* 16×16 *tiles (1024 tiles in total) decomposing a* 512×512 *image, using either a single layer (red line) — one compositing agent with many rendering agents — or a dual layer of compositing agents (dashed blue line).*

that the serial fraction strictly corresponds to the overhead as we normalized the numbers.

Nodes	Overhead (ms)	Normalised time (ms)	Speedup	Efficiency	Serial fraction
1	71.711	1000.0	1.0	1.0	—
2	76.618	540.763	1.849	0.925	0.08153
3	90.779	400.208	2.499	0.833	0.10031
4	96.114	328.187	3.047	0.762	0.10425
5	94.203	279.86	3.573	0.715	0.09983
6	92.84	247.555	4.04	0.673	0.09707
7	93.818	226.431	4.416	0.631	0.0975
8	101.466	217.502	4.598	0.575	0.10572
9	116.632	219.775	4.55	0.506	0.12225
10	112.002	204.831	4.882	0.488	0.11648
11	115.534	199.923	5.002	0.455	0.11992
12	119.221	196.579	5.087	0.424	0.12354
13	119.512	190.919	5.238	0.403	0.1235
1	30.667	1000.0	1.0	1.0	—
2	32.788	517.454	1.933	0.966	0.03491
3	41.277	364.388	2.744	0.915	0.04658
4	48.096	290.429	3.443	0.861	0.05391
5	48.798	242.665	4.121	0.824	0.05333
6	45.235	206.79	4.836	0.806	0.04815
7	45.553	184.03	5.434	0.776	0.04803
8	51.665	172.832	5.786	0.723	0.05466
9	49.262	156.965	6.371	0.708	0.05159
10	48.568	145.501	6.873	0.687	0.05056
11	46.948	135.069	7.404	0.673	0.04858
12	51.925	132.702	7.536	0.628	0.05386
13	51.699	126.263	7.92	0.609	0.05345

Table 5.2: *Performance analysis of a 512×512 image divided into 1024 16×16 tiles using a single composition layer (first part of the table) and a two composition layer system (second part of the table)*

5.6 Conclusion

We presented in this chapter a distributed graphic pipeline implemented on top of the multi-agent system described in the previous chapter.

As a distributed pipeline, its principal differentiating factor from similar architec-

tures (*e.g.* distributed visualisation pipelines based on streams such as Chromium [130]) is the use of software agents for the pipeline components, allowing the system to allocate components at run-time, on-demand, rather than using static allocation.

Performances of the pipeline are good, using an open low-latency communication protocol based on PLIST or binary representation. The possibility to broadcast messages allowed us to implement non-centralized mechanisms.

The system scales reasonably well with the number of nodes at our disposal, reaching 20 fps with the visible human dataset ($1877 \times 512 \times 330$) using DSR agents. As such, the system allows fast enough distributed volume rendering to be useful and relevant, but contrary to other architectures retains all its flexibility. The cluster of machines we used is connected through a standard, switched gigabit Ethernet network, not optimised for visualisation tasks.

The designed simplicity of the outside interfaces allowed us to create many different clients, most noticeably a PDA control client (allowing users to use their PDA device as a remote control for the visualisation), a generic fat-client viewer displaying real-time graphs in parallel with the visualisation, and a web user interface, allowing flexible user interface creation.

PART III

An Agent-based Reflective System

(CHAPTER ...6)

The Reflective Pipeline

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

— Donald Knuth

Contents

A COMMON APPROACH to system optimisation consists into having programmers working at a certain abstraction level (through the abstractions offered by a programming language) and having to intervene at lower levels to optimise the system. A typical example is to write programs in a high level language while writing critical sections of the code in a lower, more efficient language. One of the problems triggered by this approach is the tight coupling introduced, which makes the code more difficult to maintain and refactor.

A different, and more powerful, approach is to work at an upper abstraction level, and redefine how a system works using its metamodel. This metamodel is by definition a model of how the system itself operates.

One important design choice system designers face is to choose how to represent the metamodel; choosing the system itself to represent it makes the system reflective,

that is, the system is described using the same abstractions used to write the system itself.

A reflective computer system as introduced by Smith [226] is a system where computation can be performed on the system itself, not only on data external to the system. What we thus call *reflection* is the ability to reason and act upon the system itself [226, 255].

Reflective systems originally came from the programming language world, with languages like Smalltalk [101] and Lisp [173, 228] exposing their metamodel to the programmer, allowing much richer interactions.

A subset of reflectivity is *introspection*, the possibility to gather information on the system at runtime. Even more mainstream programming languages such as C++, Java or C# now provide at least some sort of introspection capability (C++ has RTTI, run-time type information, Java provides a reflective api via java.lang.reflect, both Java and C# provide annotations).

In the following sections, we will describe the reflective capabilities built into our system and some examples of how these can be used to improve performance, add new capabilities, or improve user interactions [25, 146].

6.1 Pipeline Model

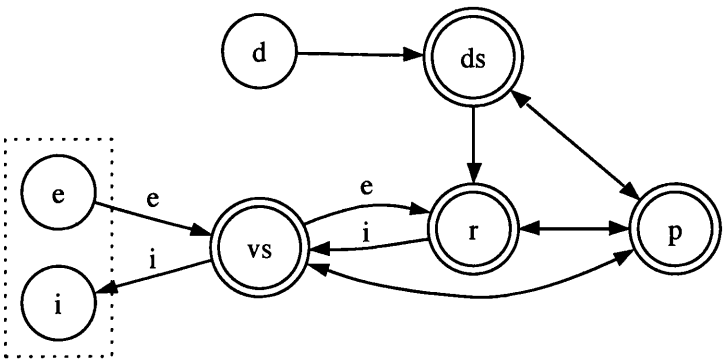


Figure 6.1: *Model of a Pipeline*

The generic pipeline we described in the previous chapter corresponds to our pipeline model. Every pipeline is composed of:

- a dataset agent (**ds**), owner of the data visualised (**d**)
- a view sink agent (**vs**), the external communication interface, which can receive requests (**e**) and returns images (**i**)
- a pipeline agent (**p**), which knows which agents participate in the pipeline

- a rendering agent (**r**), using the dataset information and requests transmitted by the view sink

Every agent knows the pipeline agent they are attached to; and the pipeline agent keeps a list of the running agents.

Agents can thus query the pipeline agent to retrieve information about the entire pipeline and the other agents that compose it.

A pipeline is considered as an oriented graph, with information flowing from one agent to the other. In addition to this, when agents specify their relationships with other agents, the cardinality can be indicated.

6.2 Storing and Accessing Information

Agents create information, and make it available to the other agents. Information gathering and information retrieval are the fundamental blocks allowing collaboration in a multi-agent system.

6.2.1 Blackboard Agent

Agents, in order to coordinate their actions, need a collaboration mechanism. One popular and well established model of collaboration is the blackboard architecture [117], where a Blackboard is a shared global database used to solve a problem. Knowledge sources respond to changes in the blackboard, and can query and modify it to solve the problem. The knowledge written on the blackboard can possibly be organized hierarchically.

The main advantage of a blackboard architecture is its flexibility; it is by nature open and can accept many clients. There are some drawbacks to the flexibility of the blackboard architecture, such as the lack of communication language (or more exactly, the communication language is constrained to the language used to represent the knowledge on the blackboard), or the possibly expensive complexity of the cooperation (as determined at runtime) [46].

As our knowledge domain is rather small, we benefit more from its flexibility than from its drawbacks.

In our implementation, blackboards are agents, providing a tuple database that other agents can access. A tuple database as a collaboration mechanism is inspired by the Linda [99, 44, 45, 32] programming language. JavaSpace [236] is an example of a similar implementation for the Java platform. Our implementation is quite similar to the JavaSpace one, with the same kind of API (though obviously we work with and use Objective-C capabilities and not Java):

- a read message, returns the content associated with a key
- a remove message, deletes the content associated with a key
- a write message, set the content associated with a key

Message content is a serialized PLIST. The blackboard agents also automatically provide serialisation – they regularly save the content of the blackboard to disk.

A tuple database as provided can be considered as a simple shared memory model. While not part of the generic pipeline model, this agent is very often added to a pipeline.

6.2.2 Relationships Statistics

In addition to information added directly by agents, in the metamodel (relationships, cardinality) or via a blackboard agent, the system can generate some metrics automatically, on demand; specifically, every connection time can be measured and statistics gathered.

This information in turn can be used by agents to determine bottlenecks and identify faulty or slow nodes.

6.3 Reflective Patterns

The system as presented and in particular the graphic pipeline allows reflective actions, *i.e.* modifying the system architecture itself. In the following sections, we will present some general patterns modifying the graphic pipeline to add new capabilities to the system. Figure 6.2 shows the conceptual model we use; a rendering operation (*r*) depends on parameters stored in the pipeline environment (*e*) to generate an image (*i*).

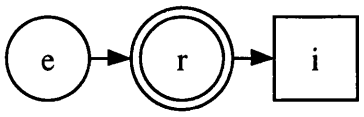


Figure 6.2: *Pipeline model*

For clarity reasons we do not show in this model the *view sink* agent, which on the running system serves as a gateway between the pipelines running on a cluster of machines and the external clients; the different requests a client can send (*e.g.* new parameters triggering an image rendering) are shown flowing directly to the concerned agents.

6.3.1 Feedback Control: Framerate Steering

An important aspect of an autonomic system is the ability to control the performance of the system. In particular, one wants to have specific goals to reach, and have the ability to modify the system to reach those goals.

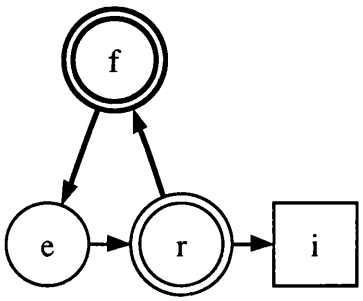


Figure 6.3: *Framerate Steering strategy*

A first step toward this is to introduce in a pipeline a feedback loop, where a given goal can be measured and the pipeline characteristics modified in real-time to approach this goal. Figure 6.3 shows how a single agent f can be added to the generic pipeline model to control the framerate. The agent is registered as a listener on the rendering agent (r) for the rendering time. The framerate agent have a specific framerate target it wants to achieve, and can modify the pipeline environment to do so, controlling its action using the rendering time.

In this example, the agent uses the rendered image size as a control characteristic to approach the goal; other agents could instead work on a different set of control parameters (choosing among different types of rendering agents, or variying the rendering quality).

6.3.2 Adding Pipeline Features: Saving Viewpoints

One application we developed is a method to save the current camera position (viewpoint) and retrieve a list of saved positions. The user can thus create a custom list of usual positions for a particular dataset. While not a fundamental example of the advantages of a reflective system (as this could easily be implemented in a static system), we implemented this functionality using separate agents that plug in an existing rendering pipeline. Figure 6.4 on the next page shows the architecture enabling this functionality.

We add three agents to the archetypal pipeline on the system side (bold agents *sp*, *db* and *vp* on the figure). One is a data agent (*db*), simply holding the viewpoint list. The second, *sp*, can add a viewpoint to the list by querying the current viewpoint

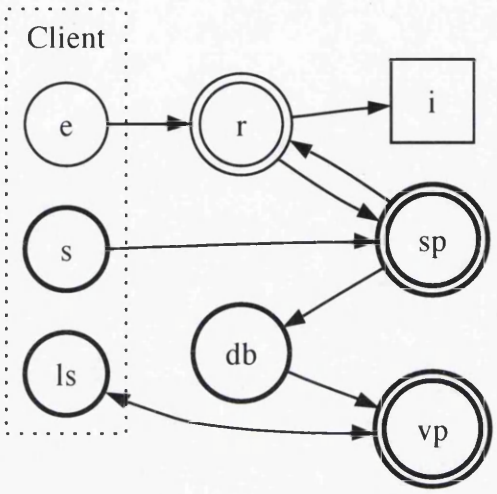


Figure 6.4: *Saving viewpoints strategy*

and setting it in the data agent. The last one, *vp* is charged to return the actual list (it can simply be an interface of a 'data' agent).

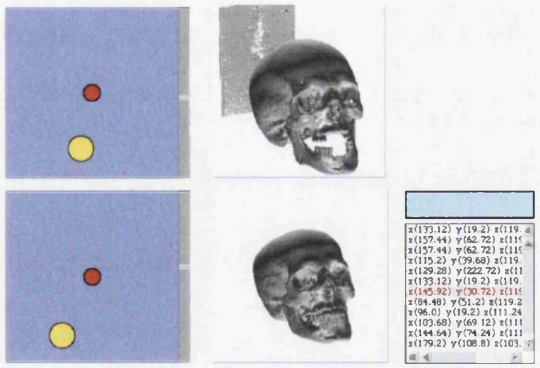


Figure 6.5: *Saving viewpoints and mediator application (Squeak prototype)*

On the user side, two additional agents are needed, one to ask the pipeline to add the current viewpoint (*s*), and another one to query the viewpoints list (*ls*). Figure 6.5 shows the prototype of this feature running in the Squeak environment.

6.3.3 Mediator Agent

Another example of an agent interacting with the pipeline to extend the functionalities of the system is a mediator agent.

With two rendering pipelines running in parallel, a mediator agent can be used to

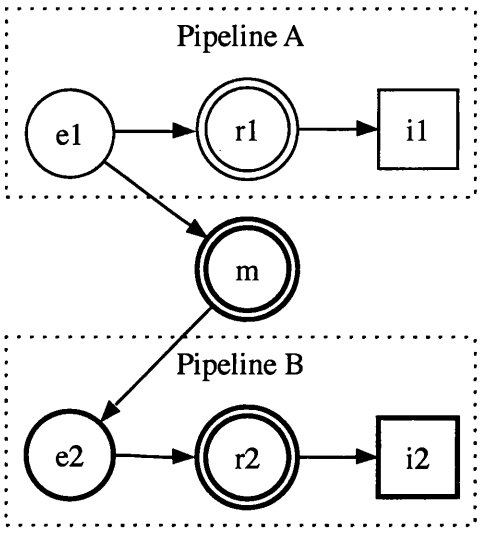


Figure 6.6: *Automatic mediation*

transform automatically some coordinates such as the camera viewpoint used to visualize one dataset into the equivalent coordinates for another dataset.

We can then manipulate one pipeline and have movements replicated on a second pipeline, in real-time after the transformation step by the mediator agent. This can be used as a bridge between two similar pipelines providing some basic collaboration mechanism (figure 6.5 on the previous page).

Figure 6.6 shows the architecture of the system. We show here two pipelines running in parallel, *e1*, *r1*, *i1* and *e2*, *r2*, *i2*, with *e1*,*e2* data containing the environment triggering the rendering of images *i1*, *i2* by the agents *r1*, *r2*. We use a mediator agent *m* to couple the pipelines' environments.

The mediator agent *m* is registered as a listener to the environment *e1* of *environment A* and knows the delta between the environments (after a calibration step). Using controls of *pipeline A* will update *environment A*; the mediator is then notified of the change, and will update *environment B*.

6.4 Resource Discovery

In an autonomous system, one key aspect is self-configuration. Autonomous systems need a way to configure themselves without outside intervention, in order to respond to any new situation.

To self-configure, it is necessary to discover new resources automatically and take

advantages of them when they appear. We implemented resource discovery on top of the basic agent discovery mechanism (see section 4.5.1 on page 87. The main difference is that the agent waiting for the resource is not necessarily the agent that will use it; the pipeline agent for instance is in charge of keeping a set of resources available and will follow the broadcast/reservation steps to do so. Another typical case is when a pool of agents are involved – the pool agent is then the one responsible to add those new resources.

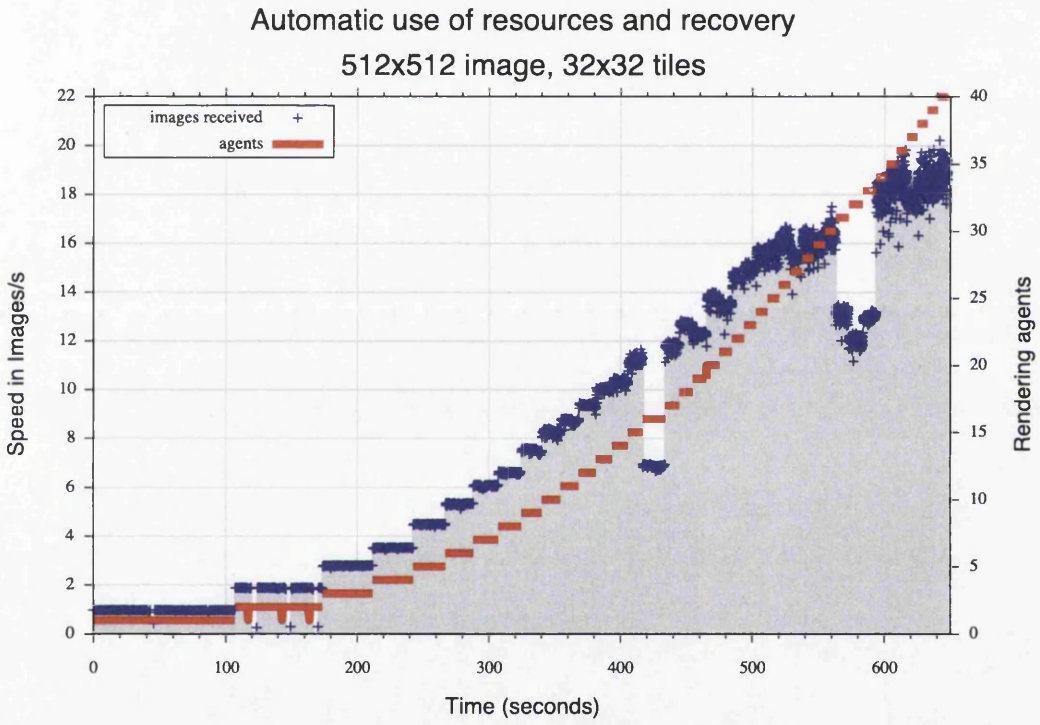
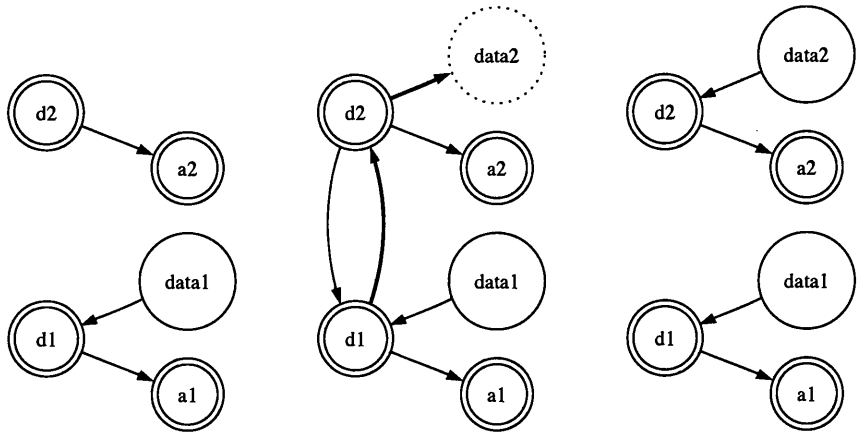


Figure 6.7: *Discovery and addition of new resources when available*

Figure 6.7 shows the performances (in images/s) reached by a distributed rendering pipeline, associated with the number of rendering agents (red lines) used. We gathered those performances data by running a distributed pipeline rendering a 512×512 image of the Visible Human dataset, using 32×32 tiles, and regularly creating via a script new rendering agents in the cluster. We can see in the graph that the pipeline discovers the availability of new resources (the rendering agents here) and automatically take advantage of their presence to improve the performances. An interesting event causing performance dips appears around 16 rendering agents used and 32 renderin agents used. We will explain in detail why this happen and how it can be corrected in section 7.7.1 on page 144.



(A) Initial state (B) Data transfer (C) Transfer complete

Figure 6.8: *Automatic data distribution*

6.5 Data Distribution

The data agent implements a simple transfer mechanism. Data agents are responsible for providing data to other agents; they have a resource store associated with them [1] where they can store data. Other agents which need particular data will send a request to their local node data agent asking for it.

If the data is already available in the data store, the data agent returns it [2]. If not, a broadcast request will be sent asking for the resource; the first agent answering will be asked to initiate a transfer connection and send the data to the data agent. Figure 6.8 illustrates this particular case. We have two sets of agents running on different nodes, with two agents *a1, a2* and two data agents *d1, d2*. *Data1* is some data requested by *a2*. As the data is not available locally, the local data agent *d2* sends a broadcast request asking for it. The data agent *d1* answers the request, connects to *d2*, and if the first in line starts transferring *data1* to a local copy *data2*.

6.6 Failure Recovery

An important aspect of an autonomous system is self-healing — the need to adapt itself to any failure and continue to work seamlessly. We describe in this section our approach to system reliability and failure recovery.

[1] In our current system, this simply corresponds to a specific directory on disk.
[2] In our current implementation, this simply corresponds to a file path.

6.6.1 REST Approach

When a failure occurs in a system, one difficulty is to recover the system in a working mode. A typical solution is to have a journal of events and replay this journal upon error to reach a stable state.

Our general approach within our system is to have agents not depending on runtime state as much as possible – following a REST approach [83] where the entire state needed to perform the computation is passed with or can be inferred upon the request (message sent to the agent).

In more practical terms, the main application of our system is the graphic pipeline; the state management for a pipeline consists into having a pipeline agent hosting a model of the pipeline (a set of agents in the pipeline along with their relationships). This state is saved on disk by the pipeline agent and synchronized, so that if the pipeline agent dies the pipeline can be restarted without problems (along with a broadcast message asking agents of a given pipeline to identify themselves).

Additional information such as rendering parameters are delegated to a blackboard agent that also saves its content on disk and can be recovered after a crash. More importantly, no state (other than which dataset is visualised) is absolutely necessary for the rendering agents to generate an image, and this additional information is therefore not primordial for the pipeline; in the worst case where rendering parameters such as the rendering quality or the camera viewpoint are lost, the rendering agents will simply use default values.

6.6.2 Pool Management

One of the strategies used in our system is to have specific agents responsible for the availability of other agents. The pool agent, for example, maintains a pool of agents available. The pipeline agent does the same for agents registered in the pipeline.

Agents only have to specify that they need a pool containing a specific kind of agent (basically, requesting a $1 - n$ relation), and the system returns them a filled pool with the available agents. This list can possibly be null if no adequate agent is running, in which case the system will automatically start agents on nodes receiving the broadcast request (if the nodes have the resources to do so).

A pool here is a local (to an agent) list of other agents; while the pool agent is only in charge of giving a list of agents to the requester, and checking that they are available (possibly notifying clients that an agent is suddenly not available).

A pool can be either *configured* or *unconfigured*; the unconfigured state means that the agent using the pool have to configure the pool's agents. When the configuration step is done, the agent indicates it to the pool, which then switch to the configured state.

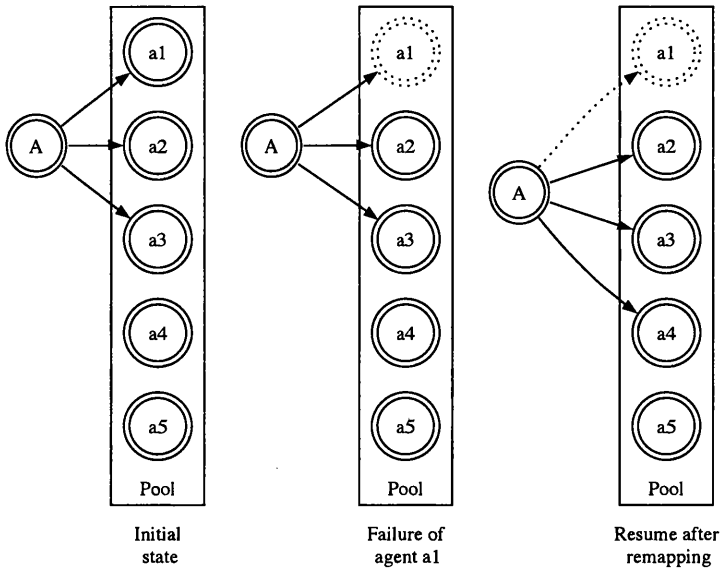


Figure 6.9: *Failure and recovery within a pool of agents*

When a failure happens, the pool switches back to the unconfigured state, prompting the agent to reconfigure it. Figure 6.9 shows that process: an agent *A* uses three agents (*a1, a2, a3*) of a pool, when one the agent fails (*a1*). The computation can resume once the agents in the pool are reconfigured.

As an example, if the agent is using the pool within Map/Reduce, the typical configuration steps involve doing (or redoing) the *split* operation using the set of agents the pool contains.

In the rendering pipeline, pools of rendering agents are used, scaling well as all the necessary information is passed during the request after the first configuration handshake.

6.6.3 Failure Detection

Failure in a communication can be detected in two ways:

1. the calling agent will catch an error and report it to the pool agent
2. the pool agent runs regular checks on its agents to proactively detect failure and warn its clients

When case *1* happens, the calling agent invalidates the failed agent in its local pool, and notifies the pool agent that a problem occured. This permits the pool agent to keep track of problems, identifying machines with high failure rate, and possibly restarting agents.

When case 2 happens, the pool agent invalidates the failed agent and warns the agent using the pool of the failure. It may be the case that the agent already identified the failure on its own, but this dual approach ensures the shortest time for failure detection.

6.6.4 Failure Examples

When a failure occurs, two possible things can happen:

- other agents in the pool are immediately available
- no agent is available

In the former case, recovery (to the same performance level) is very fast, while in the latter recovery entirely depends on the starting time of the used agents.

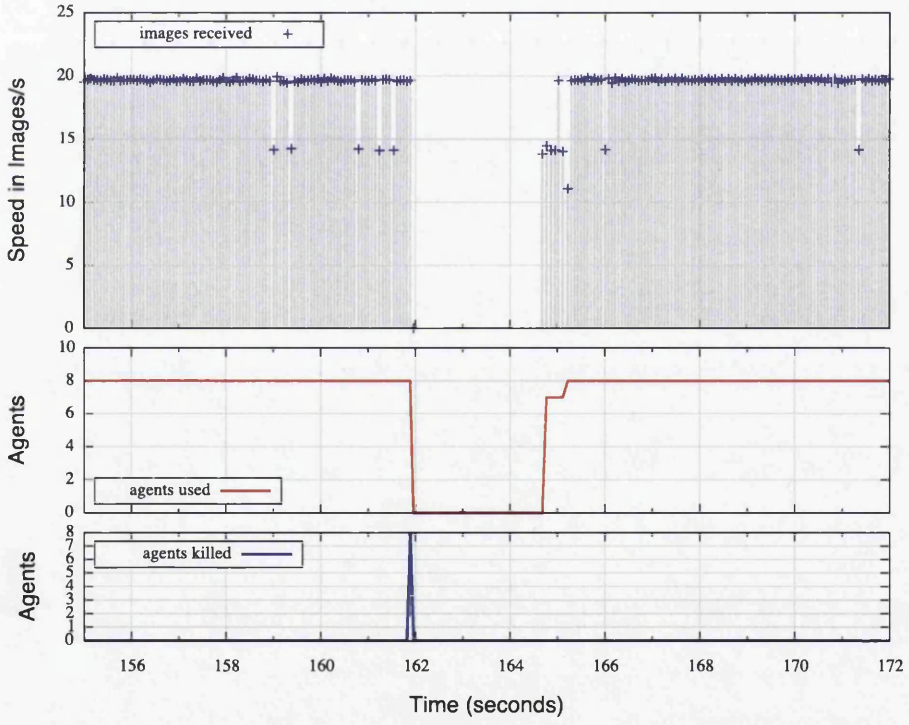


Figure 6.10: *Failure and recovery with a pool of 16 agents, working on a* 512 × 512 *image divided into* 64 *tiles*

Figure 6.10 shows a failure occuring on a distributed rendering pipeline, generating 512 × 512 images of the 600 MB Visible Human Dataset using 8 remote agents, at 20 frames per second.

As can be seen when the failure occurs (just before second 162), as long as agents are immediately available the recovery is very quick (in this example 2.5 seconds, when

we kill all the remote rendering agents used by the pipeline), and after the recovery we are getting the same performance levels as previously reached.

The red curve shows the number of agents used by the pipeline; when the first image after the failure reaches the client (just before second 165), only seven agents (out of the 8 asked) are ready and can be used. A few frames later the last agent becomes available and is thus automatically used by the system, which then returns to the original level of performance.

Figure 6.11 shows on the other hand what happens when no available agents are present in the pool; the performance drops, and returns to a level depending on the number of remaining agents in the pool. Recovery starts automatically, but is delayed by the starting time (4-5 seconds) of the rendering agents used here.

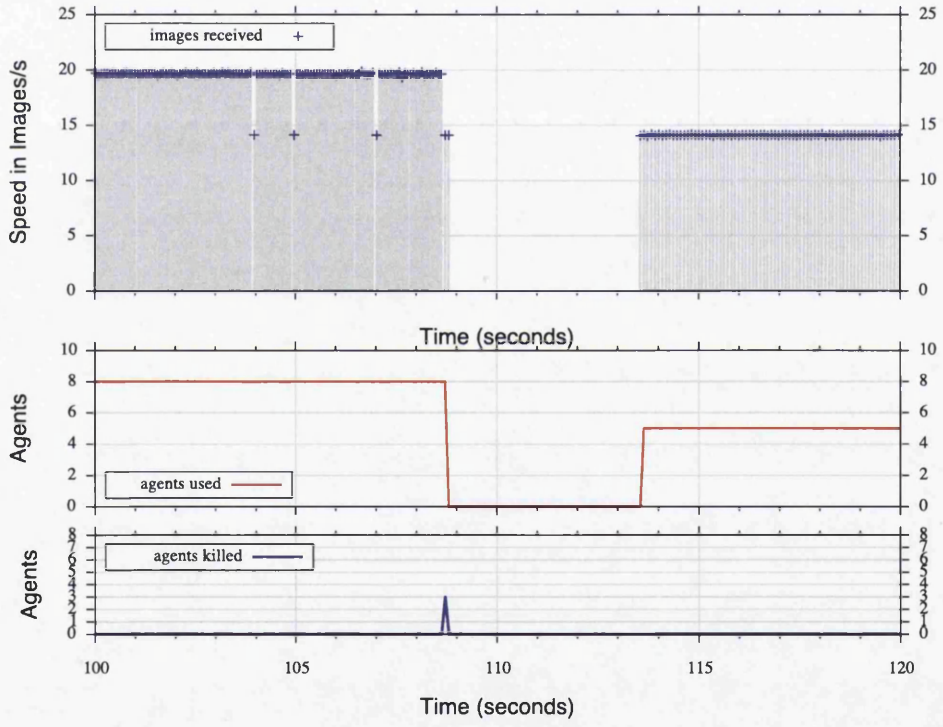


Figure 6.11: *Failure and recovery with backup agents*

6.7 Conclusion

We presented in this chapter the reflective architecture of our system, and how it is possible to take advantage of the same mechanisms used to implement the distributed graphic pipeline to do so. Mechanisms such as the blackboard system, used by agents to exchange informatino and collaborate, were presented.

We described some reflective patterns implementing autonomic behaviour, such as an automatic framerate steering capability, using an agent acting as a feedback control on a pipeline. We also described important mechanisms toward self-management, namely the resource discover mechanism and the failure recovery management architecture, leveraging a REST approach and specific architecture such as pools of agents. Finally, we presented experimental examples of failure recovery.

The different patterns described present various examples of modifications of a system to extend its capabilities; this architecture allows the implementation of autonomic features such as self-configuration and discovery, failure detection and self-healing.

(CHAPTER ...7)

Visualisation Strategies

"Will it be possible then, for people to say stonily, that poems are not real, and that patterns are nothing but images; when in fact, the world of images controls the world of matter"

— Christopher Alexander

Contents

A VISUALISATION STRATEGY [1] is a specific visualisation pattern, such as splitting the viewing output into different images, or distributing a rendering, that we identify and isolate. By using a generic notation, we can describe those patterns in a reusable way; and by implementing those same patterns as agents, we can achieve reuse in our system, as agents filling the same role (*e.g.* a rendering agent) can be transparently replaced by another.

[1] The work presented in this chapter was partially published in [212]

7.1 Genericity and Composition of Strategies

Looking back at the general architecture shown in fig. 5.1 on page 98, we can simplify it by considering only the parameters needed to generate an image, the rendering agent itself, and the final result (fig. 7.1, also see section 6.3 on page 122).

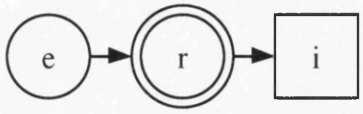


Figure 7.1: *Generic Pipeline*

In general, strategies do not provide a rendering implementation themselves, but delegate the rendering to existing rendering agents. Strategies can be considered both as a type of software pattern, organising existing agents according to a predefined architectural pattern, and as abstract data types: as long as an agent implements the same interface as the one used in a strategy description, it can be used transparently in (or as) a strategy; which means a given strategy can be combined with any of the available rendering algorithms.

Conversely, we can say that strategies are generic behaviours: a new rendering algorithm will be able to transparently take advantage of the existing strategies in the system.

As strategies respond to the rendering protocol, they are considered by the system as normal rendering agents, and can thus be composited as well: in a given strategy, instead of a 'true' rendering agent, another strategy can be used.

The following sections demonstrate some of the strategies we created.

7.2 Progressive Rendering

Figure 7.2: *Progressive rendering*

Progressive rendering [156] is a mechanism that computes a rendering in a low resolution, then gradually increments the resolution to improve the quality. Fig. 7.2 shows an example of a three-step progressive rendering for a volume dataset.

Although progressive rendering by rendering each separate step is inherently a longer process to get the final image than rendering the final image immediately (discarding for the moment any possible parallelisation of this process), it is a very useful mechanism, as it allows the user to have a quick feedback of what will be the final result. Moreover, the low quality rendering can then be fast enough to achieve interactive frame rates with minimal resource consumption.

It is also possible to implement progressive rendering in such a way that it would not bring download of full images, using progressive JPEG and reorganising JPEG blocks. We did not implement this as the bottleneck with visualising a volumetric dataset is the rendering itself, not the image download (see [55] for simulation experiments using e-Viz's SimuVis).

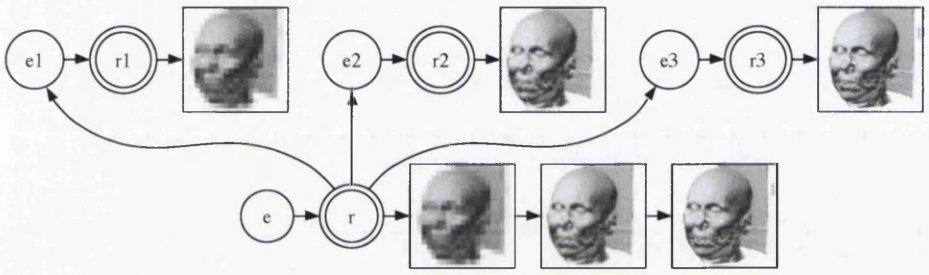


Figure 7.3: *Progressive rendering agent*

To create a progressive rendering strategy, we use an agent answering the rendering protocol, which will get the environment, and where visualisation clients can connect to. Figure 7.3 shows the architecture.

This agent implements a three-step progressive rendering, and therefore needs three rendering agents. When receiving a rendering request, it modifies the resolution requested, and passes it to the rendering agents. The progressive rendering agent simply acts as a view sink to each of the rendering agents. When it gets a result (*i.e.*, an image), it forwards it to the 'real' view sink.

7.3 Distributed Rendering Strategy

Distributed rendering is the action of generating parcels of a rendering on different computers, then creating a final result from these parcels.

Broadly, two approaches exist:

1. image-space distribution
2. object-space distribution

In image-space distribution, we work on the final image – splitting the image into different areas, with each area rendered on a different computer. Figure 7.4 on the

next page shows how rendering agents *r1, r2, r3, r4* read data *d*, then render partial images. A compositor agent *c* uses those partial images to create the final image.

This method works well when the rendered data is not the bottleneck, e.g. when you can easily distribute the original data, as every agent uses the same dataset (though the actual datasets could be replicated to improve latency). The principal advantage of image-space distribution is that it permits to parallelise the rendering easily, as modifying a rendering algorithm to render a partial image is a minor modification.

Conversely, Object-space distribution (Figure 7.5 on the following page) works on the objects (dataset). The typical usecase of this method is when we try to render a very large dataset; by splitting the original dataset into smaller sub-datasets, and rendering intermediate images that are then recomposited, we can work with larger datasets than a single node could cope with. Figure 7.5 on the next page shows how the original dataset *d* is split into smaller datasets *d1, d2, d3, d4, d5, d6, d7, d8*. Each dataset is rendered by agents *r1, r2, r3, r4, r5, r6, r7, r8*. The final image can then be recomposited from those partial images by a compositor agent *c*.

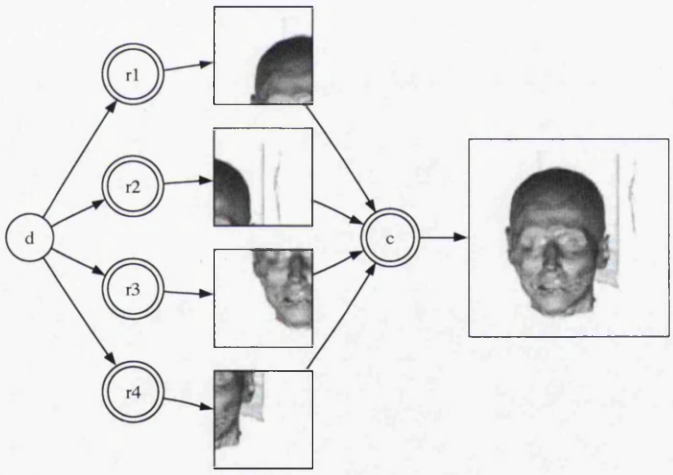


Figure 7.4: *Image Space rendering*

The best approach is ideally a mix of these two methods, although image-space distribution works extremely well with parallelisable rendering algorithms like ray-tracing and ray-casting, as each ray is independent from another.

The distributed rendering mechanism we choose to implement is one based on image-space, as it allows good scalability as well as being adaptable to any type of renderer (as long as the renderer can generate a 2D image).

The general architecture of this visualisation strategy is rather close to the one implemented for the progressive rendering agent (figure 7.3 on the preceding page).

The main agent implementing this strategy needs to reserve a number of rendering agents; the original environment is modified, used by the rendering agents to render

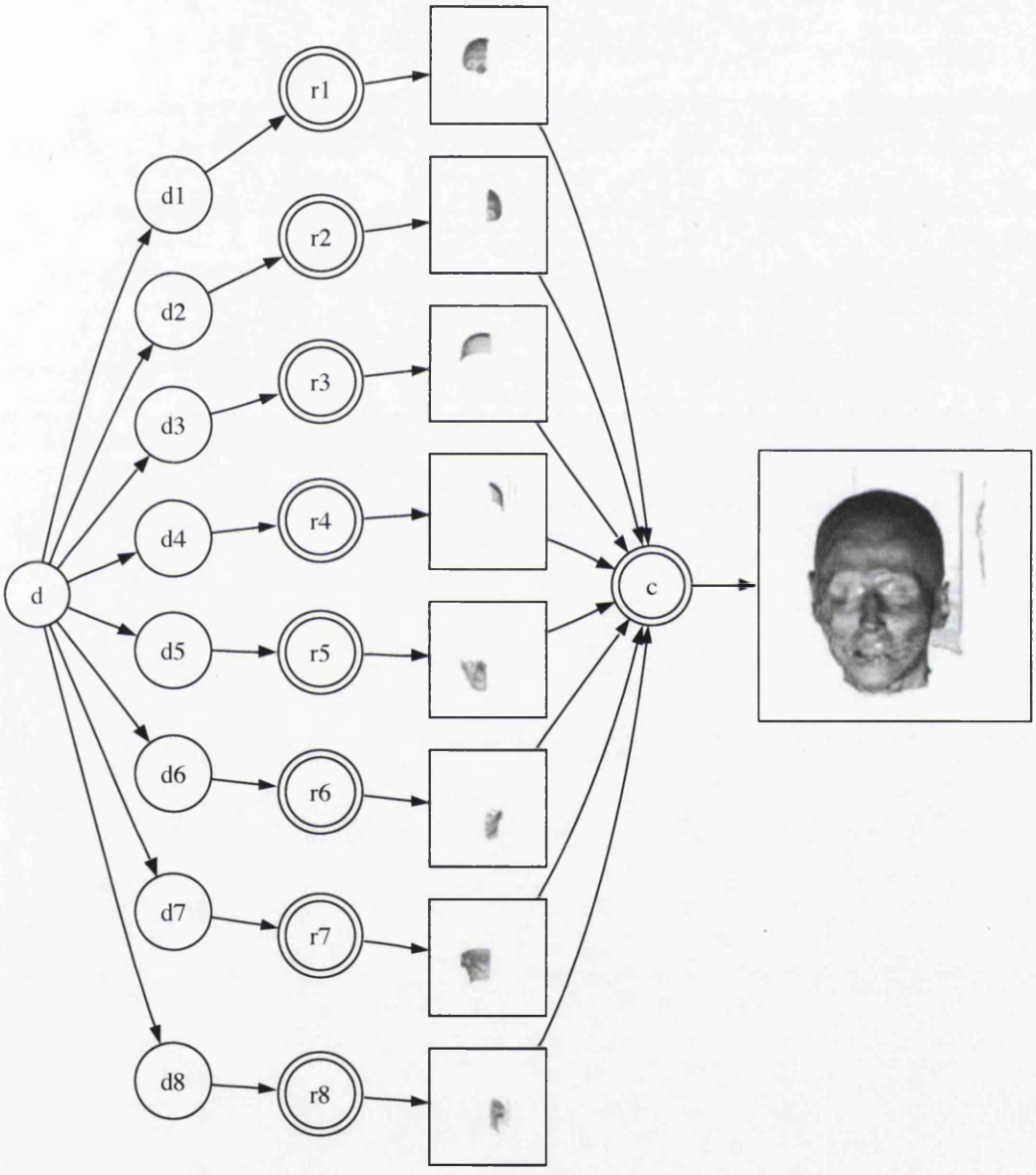


Figure 7.5: *Object Space rendering*

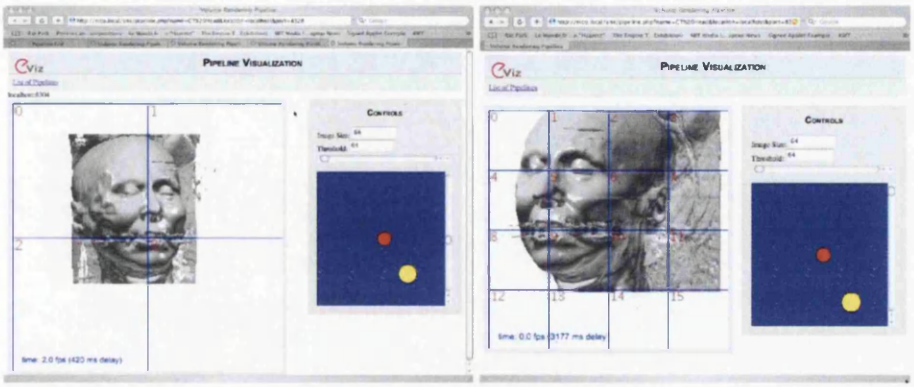


Figure 7.6: *Screenshots of a parallel image-space rendering pipeline*

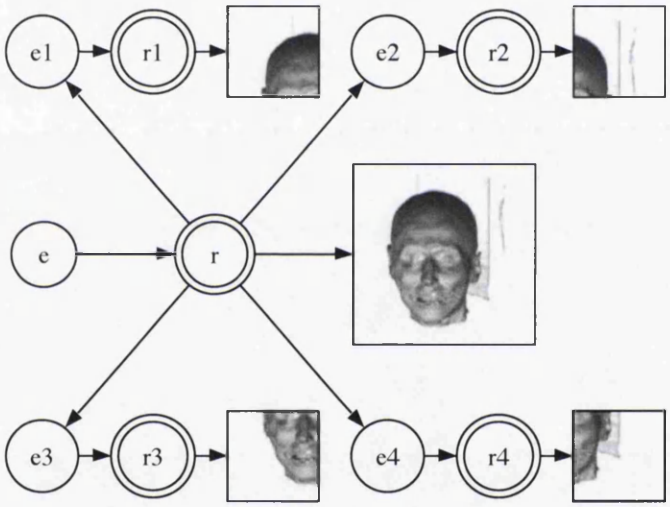


Figure 7.7: *Distributed rendering strategy*

partial images.

The target image (the image we want to render) is divided into tiles. The tiles are split among the available rendering agents, and for each tile a corresponding viewpoint (the camera position) and look-at point (where the camera points to) is computed using the original viewpoint/look-at point couple.

The rendering agents do not need any kind of modification; they will simply receive a modified environment using the new viewpoints corresponding to the tile they need to render.

Every tile is returned to the main agent on completion; the agent recomposes a complete image using those tiles, which can then be sent back to the view sink agent of the pipeline.

We also implemented in our system an optimised distributed rendering using the

Map/Reduce implementation (see section 7.7 on page 143), which requires the rendering agents to know how to compute specific viewpoints themselves, but performs better.

7.4 Tiled Display Strategy

Another strategy is rather close to the above distributed image-space method. What we want to achieve is a tiled display, that is various screens mounted as tiles to form a giant display.

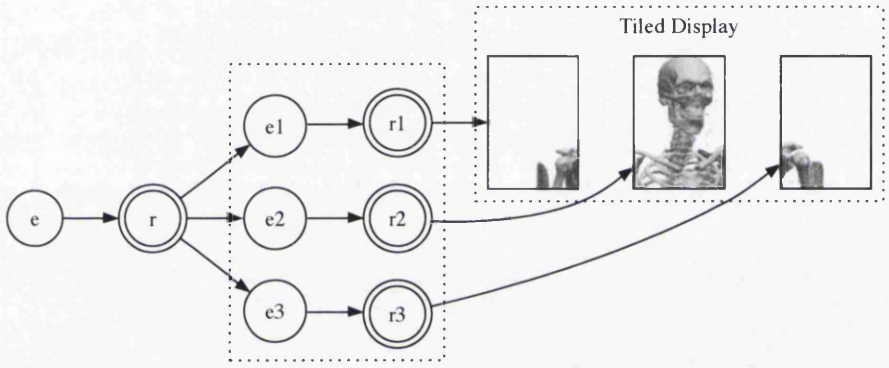


Figure 7.8: *Tiled display strategy*

The mechanism is exactly the same as the distributed rendering strategy, with few differences:

- Instead of a single image returned to a single view sink, this agent returns as many images as specified for the tile display, to the corresponding view sinks
- no image is recomposited using the partial images, as those are sent directly to the view sinks
- the final tiled image is of course of much higher resolution than on a usual pipeline

As we highlighted before, strategies, being generic, can be easily composed together. Figure 7.9 on the following page shows an example of using progressive rendering agents as rendering agents of a tiled display. An original tile pipeline *e1, r1, i1 / e2, r2, i2 / e3, r3, i3*, with *e1, e2, e3* representing the environment used by the rendering, *r1, r2, r3* the rendering agents and *i1, i2, i3* the rendered tiled images is used. The rendering agents *r1, r2, r3* are replaced by progressive rendering pipelines, so that for example $r1 : e1_a \mapsto r1_a \mapsto i1_{64}$, $r1 : e1_b \mapsto r1_b \mapsto i1_{256}$, $r1 : e1_c \mapsto r1_c \mapsto i1_{512}$.

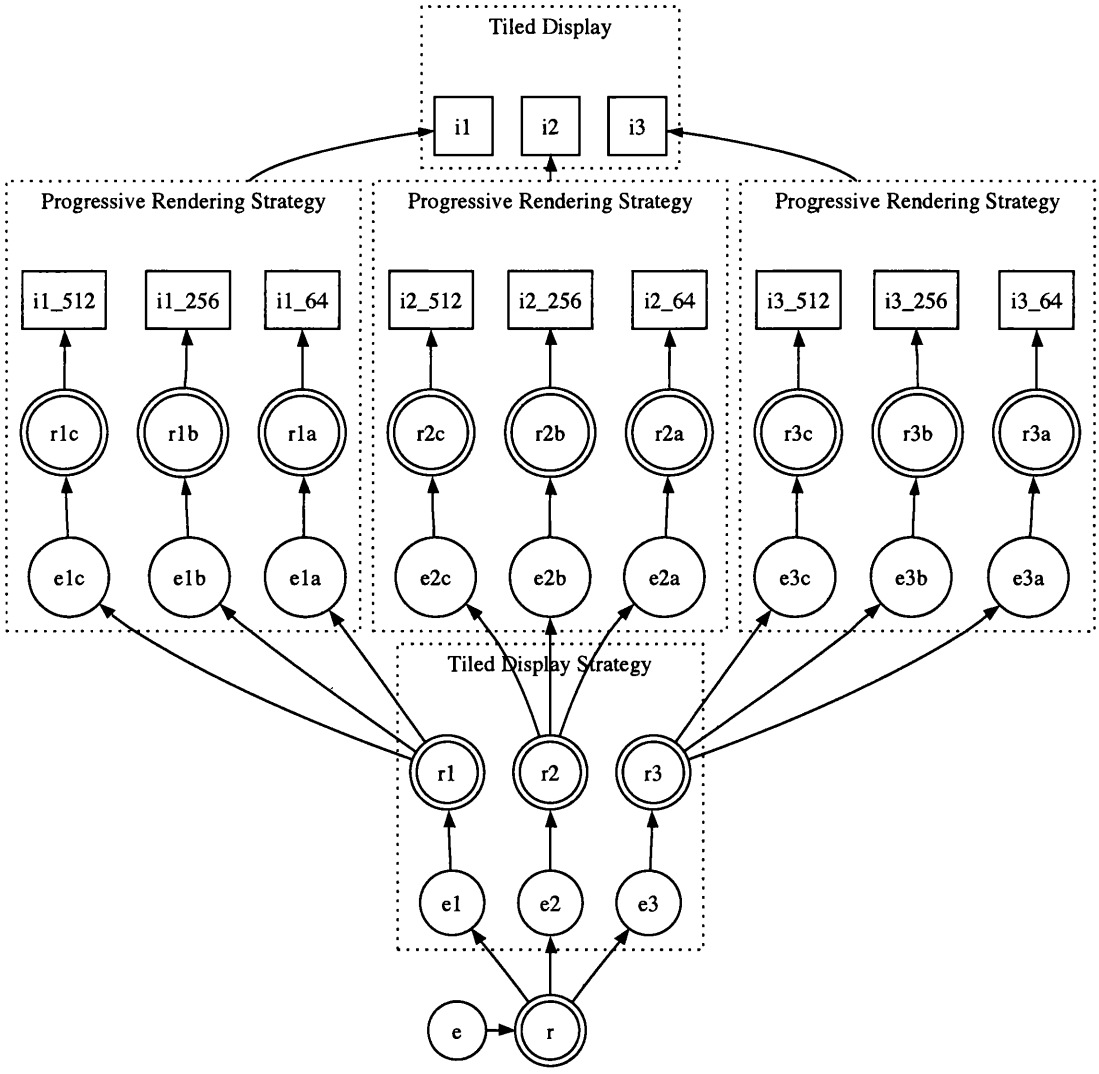


Figure 7.9: *Composition of strategies*

7.5 Adaptive Subdivision Strategy

A type of framerate steering (see section 6.3 on page 123), the adaptive subdivision strategy works for image-space distributed rendering. An agent implementing that

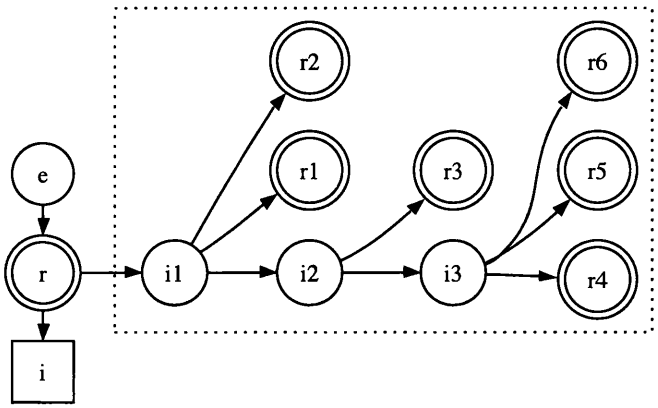


Figure 7.10: *Adaptive Subdivision Strategy*

strategy will divide the rendered image into non-regular grid, so that the rendering time is averaged across the available rendering agents. This non-regular grid is built using a serie of regular grids with more and more tiles. Those regular grids (figure 7.11 on the following page) are aranged in a layered manner, with the grids at depth $n + 1$ being more refined as the one at depth n. An important arrangement is that a tile at depth n exactly corresponds to a set smaller tiles at depth $n + 1$, as seen in figure 7.11 on the next page. The mechanism could be repeated as much as necessary.

In our implementation, we limited ourself to three levels of grids, as the advantages of distributing smaller tiles with more levels were offset by communication and synchronization delays. Figure 7.10 shows how the agent implementing the strategy (*r*) keeps a multi-level representation of the image (*i1*, *i2*, *i3*, figure 7.11 on the next page) associated with rendering agents (*r1*, *r2*, *r3*, *r4*, *r5*, *r6*).

Figure 7.12 on the following page shows a screenshot of a running pipeline, using the visible human dataset. As can be seen, the final grid is not regular, and automatically adapt to the workload from frame to frame, using the previous frame rendering time information to decide to subdivize the grid or not.

Figure 7.13 on page 143 shows the impact on performance. The sinusoidal shape is the consequence of following a basic scenario with the camera, where we rotate the visualised dataset while zooming in and out. As we see on the graph, the sinusoid has smaller amplitude using the adaptive framerate agent, which to the user provide a more constant framerate (users are particularly sensitive to varying framerates, more so than the actual framerate; a constant slower framerate would for instance

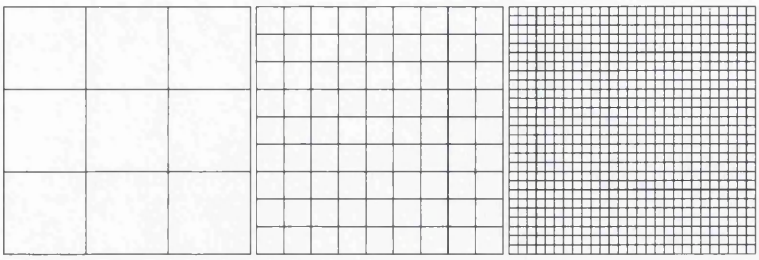


Figure 7.11: *Adaptive subdivision levels*

be preferred over a faster framerate that sometimes stops abruptly).

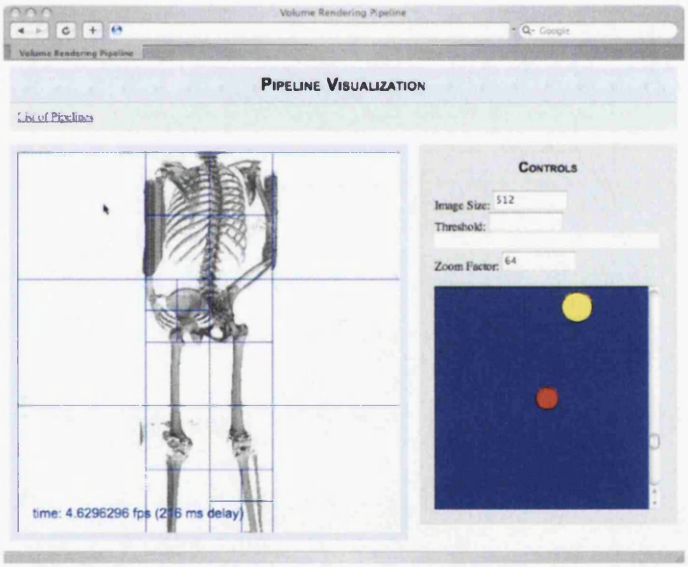


Figure 7.12: *Adaptive subdivision strategy*

7.6 Merging Pipelines

This graphical strategy allows to merge the output (*i.e.* the rendered images) of two pipelines. This could be for multiple reasons – comparison of two different datasets, or rendering the same dataset with different parameters (*e.g.* for a volume dataset, we could want to combine the output of two renderings with different isosurfaces, as illustrated by figure 7.14 on page 144).

Figure 7.15 on page 145 shows the architecture needed for this action. Two environments (*e1*,*e2*) are prepared by a merge strategy component (*m*). Two pipelines are

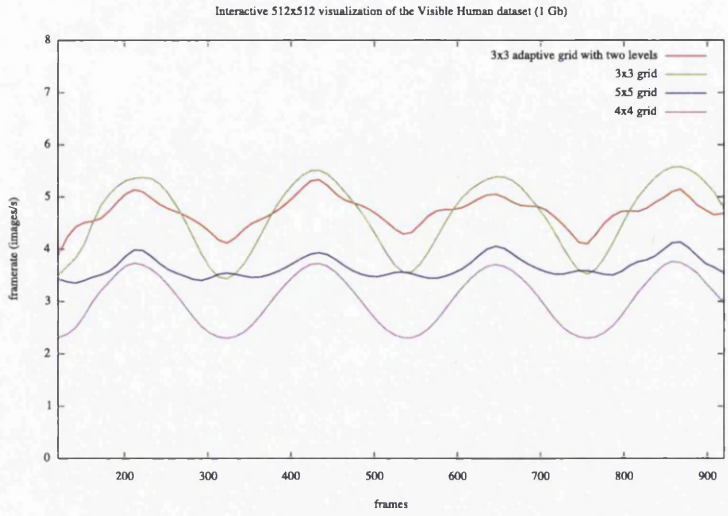


Figure 7.13: *Adaptive subdivision – impact on performance*

then run in parallel (*e1-d1-r1-i1* and *e2-d2-r2-i2*, where *d1* and *d2* are the datasets used by the pipelines, with *d2* possibly the same as *d1*). The rendered images from each pipeline, *i1* and *i2*, are composed by a compositing agent *c* to generate the final image *i*. This compositing agent *c* can be controlled on the client side (*e.g.* with *t* setting the alpha used for the composition).

7.7 Applying Map/Reduce to Distributed Rendering

Map/Reduce (presented in section 2.2.6.2 on page 30) is provided in our system to allow easier organisation of distributed computation among agents.

We used Map/Reduce as the underlying architecture to organise the distributed image rendering as presented in section 7.3 on page 135; the general principle is the same, we divide an image into a set of tiles that are then rendered by a pool of agents running on different computers.

Figure 7.16 on page 146 shows the general architecture of this organisation. An agent *a* receives an environment *e* triggering a rendering; the three different parts of Map/Reduce then take action (in bold on the diagram). The splitting operation occurs in *s* and takes a set of tiles (*0*, *1*, *2*, *3*, *4*, *5*, *6*, *7*). Splits (*s1*, *s2*, *s3*, *s4*, *s5*) are generated for each available agent *a1*, *a2*, *a3*, *a4*, *a5* used in the rendering:

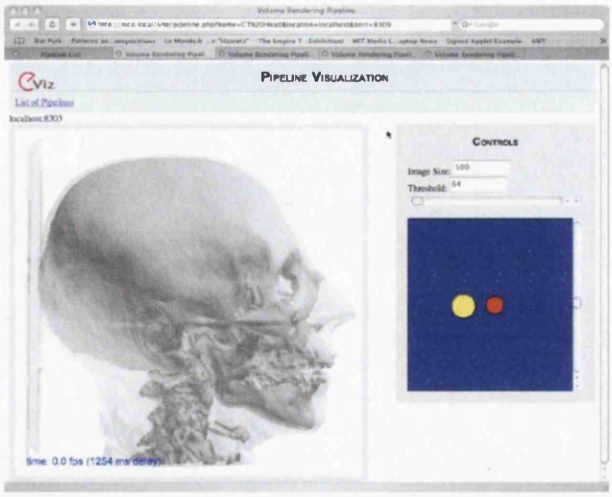


Figure 7.14: *Screenshot of a merged pipeline: we combine two basic renderers using the same dataset, with different isosurfaces (skin and bone densities)*

$$\begin{pmatrix} 0 \\ 1 \\ \dots \\ 7 \end{pmatrix} \xrightarrow{split} \begin{cases} s_1 : & (0,1) \\ s_2 : & (2,3) \\ s_3 : & (4,5) \\ s_4 : & (6) \\ s_5 : & (7) \end{cases}$$

The mapping operation m then sends the splits to the agents, which render partial images that are then sent back to the Map/Reduce agent a and recomposited by the reduce operation r to create the final image i.

The principal difference between using the Map/Reduce paradigm and the distributed rendering strategy described previously is in identifying and opening some parts of the distribution mechanism to the programmer; particularly, the programmer can now specify and change the splitting mechanism (*i.e.* choosing which rendering agent will get which part of the image to render, as shown by example listings 7.17 on page 147 and 7.18 on page 148).

7.7.1 Performance Issues and Technical Choices

The principal difficulty of applying Map/Reduce to real-time image rendering is how to minimize overhead as much as possible in order to increase the scalability of the system.

As an example of such measures, partial images are not compressed, and the bytes representing the images are directly sent on the network, sacrifying bandwidth to maximize CPU load. Removing intermediate compression on a 512×512, ARGB

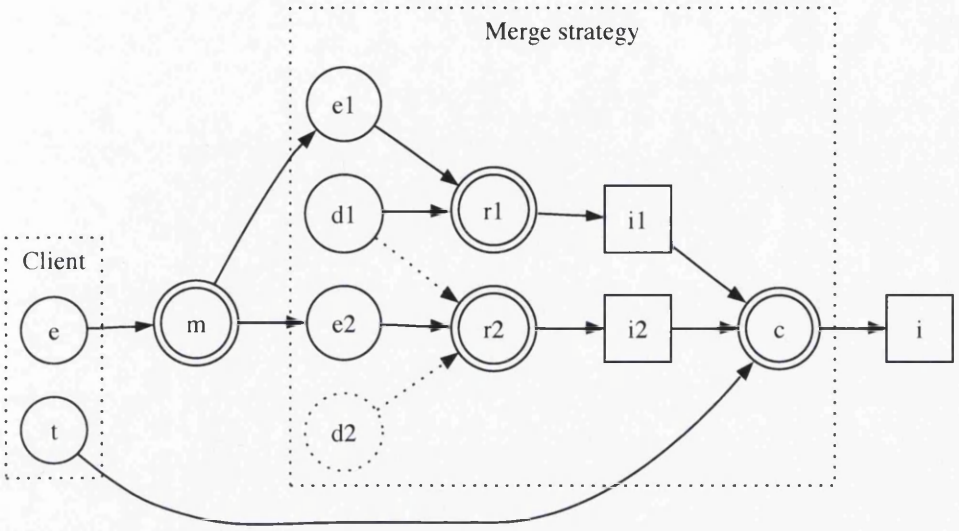


Figure 7.15: *Merging architecture*

image, resulted in a performance increase of 4-5 frames per second (a final framerate of around 20 FPS), using a decomposition of the image among 256 tiles of 32×32 pixels. The final image is then compressed, but by the view sink agent rather than by the agent using Map/Reduce – so that this final compression step itself does not slow down the *reduce* operation.

Another important design choice was to aggressively cache computations such as the splitting operation. Since the splitting operation is a non-critical one (it is executed only in case of a fault or when the pool of available agents change), we were able to implement it directly in Smalltalk rather than in Objective-C, which let us experiment with different splitting strategies easily (see two examples of such splitting strategies, in listings 7.1 on page 147 and 7.2 on page 148).

Many operations are done in place when possible, to minimize memory recopy operations. Communications were switched from PLIST to binary for increased performance, lowering the overhead of the complete mechanism to around 20 ms (the minimum overhead with PLIST was around 33 ms).

Figure 6.7 on page 126 shows an example of the performance achieved with that method while rendering the visual human dataset ($1877 \times 512 \times 512$, around 600 MB). Nodes are added regularly to the system (red curve on the graph) and automatically incorporated to accelerate the rendering. While open to the programmer and very flexible, this distribution method is still particularly efficient and permits to achieve high framerate.

An interesting observation is the performance dips seen around 16 and 32 rendering agents; the reason for this is that the load on each tile is averaged using the alternate

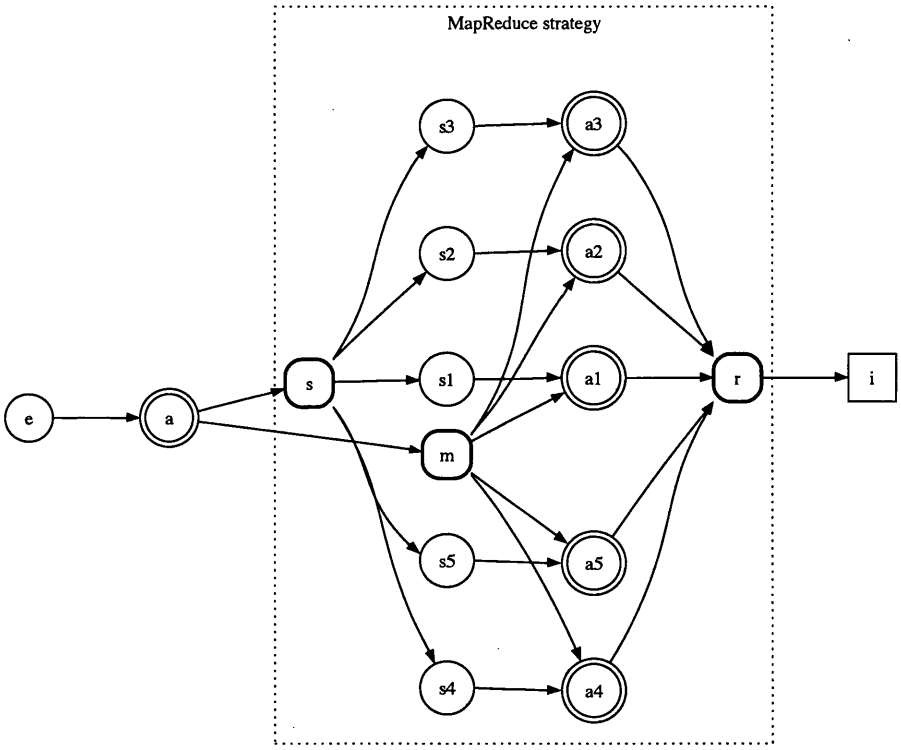


Figure 7.16: *Map/Reduce application: distributed rendering*

```
split: nodes with: tiles

nbNodes ← nodes count.
perRun ← (tiles count) / nbNodes.
perRun ← perRun intValue.
base ← nbNodes * perRun.

currentNode ← 0.
added ← 0.

node ← nodes objectAtIndex: currentNode.

tiles do: [:tile|
  (node run count ≤ perRun) ifFalse: [
    currentNode ← (currentNode + 1) modulo: nbNodes.
    node ← nodes objectAtIndex: currentNode.
  ].
  node addObject: tile.
  added ← added + 1.
  (added > base) ifTrue: [
    perRun ← perRun - 1.
  ].
].
```

Listing 7.1: Split linearly a list of tiles T_0^i in multiple runs for a pool of nodes N_0^j.

$$\begin{pmatrix} 0 \\ 1 \\ \dots \\ 15 \end{pmatrix} \xrightarrow{split} \begin{cases} n_0: & (0,1,2,3,4,5) \\ n_1: & (6,7,8,9,10,11) \\ n_2: & (12,13,14,15) \end{cases}$$

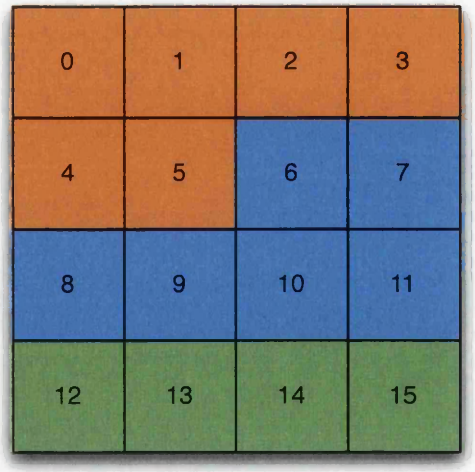


Figure 7.17: *Example of the linear split algorithm shown in the listing 7.1. We split the tiles T_0^i on the nodes N_0^j with $i = 15$ and $j = 2$ such as $T_0^{15} = (0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)$ and $N_0^2 = (n_0, n_1, n_2)$*

```
split: nodes with: tiles

nbNodes ← nodes count.
perRun ← (tiles count) / nbNodes.
perRun ← perRun intValue.

currentNode ← 0.

tiles do: [:tile|
  node ← nodes objectAtIndex: currentNode.
  node addObject: tile.
  currentNode ← (currentNode + 1) modulo: nbNodes.
].
```

Listing 7.2: Split alternately a list of tiles T_0^i in multiple runs for a pool of nodes N_0^j.

$$\begin{pmatrix} 0 \\ 1 \\ \dots \\ 15 \end{pmatrix} \xrightarrow{split} \begin{cases} n_0 : (0,3,6,9,12,5) \\ n_1 : (1,4,7,10,13) \\ n_2 : (2,5,8,11,14) \end{cases}$$

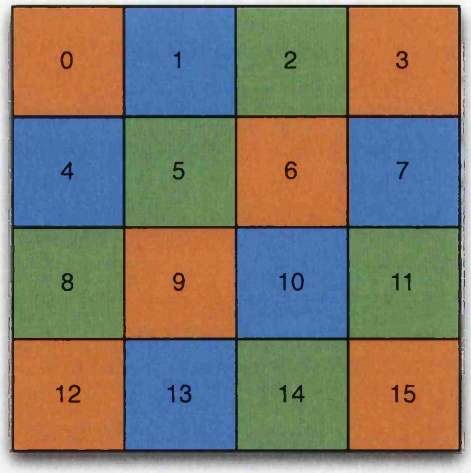


Figure 7.18: *Example of the alternate split algorithm shown on the listing 7.2. We split the tiles T_0^i on the nodes N_0^j with $i = 15$ and $j = 2$ such as $T_0^{15} = (0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)$ and $N_0^2 = (n_0, n_1, n_2)$*

split method (which gives us the best results on average), but when using a multiple of the number of tiles used, we end up having complete vertical rows assigned to the same agent (on this example we used 32×32 tiles on a 512×512 image, therefore we have 16 columns).

An important feature of the system illustrated while developing the Map/Reduce distributed rendering example is the possibility for developers to start with a naive implementation and gradually optimise the operations (by doing in-memory placement, switching later on to a raw binary communication protocol instead of the already lean PLIST, etc.), thereby improving performance without sacrifying the general openness and clarity of the architecture.

7.8 Conclusion

We presented in this chapter various examples of visualisation strategies, illustrating and taking advantage of the reflective nature of the system. Those strategies can be considered as visualisation patterns (in the software pattern sense), identifying reusable and recursive architectures. Different rendering algorithm implementations can be used interchangeably with any of those strategies, and strategies themselves can be combined to form complex systems.

This ability to interchange different rendering techniques or different algorithms answers one of the visual supercomputing needs, allowing to integrate various existing approaches, choosing the best one at runtime.

We showed an implementation of distributed rendering using the Map/Reduce paradigm, exposing more internals of the system, and thus giving more control to the programmers.

This implementation takes advantage of many features of our system to provide robustness and self-configuration, as well as staying particularly flexible and open to the programmer. We took advantage of this flexibility to gradually optimise the system, and while the implementation is now highly optimised, scales well and provides good results, it is notable that no flexibility was lost in the process, the optimisation fitting into the current architecture.

(CHAPTER ...8)

System Simulation and Reflectivity

"With four parameters I can fit an elephant, and with five I can make him wiggle his trunk."

— John von Neumann

Contents

A REFLECTIVE SYSTEM allows applications using the system to interrogate and possibly modify its behaviour at run-time. This flexibility allows the system to react to changing conditions without the need to stop or restart the entire system, thereby minimizing downtime and improving performance.

This flexibility has a cost, as any modifications to a running system have an impact; in a distributed system such as ours, changes need to be propagated to all the agents involved. If we take as an example our distributed image rendering mechanism, we can implement a tight feedback loop, with an agent monitoring the total rendering time and modifying parameters of the pipeline to improve performance (see section 6.3 on page 123). In an ideal system with instantaneous communications, such a system would be able to modify the parameters at every frame, and in return

be extremely reactive to condition changes. In a real system though, it is obvious that we cannot do that, as the impact of the changes would be minimal compared to the delay introduced by propagating those changes to all agents involved. There is thus a balance to strike between reactivity to changes and the impact of changing the system at runtime.

8.1 The Need for Modelling

From this perspective, the best possible way to determine the right parameters as well as the proper time to make changes would be to know in advance the performance of a system. One possible solution is to record past performance and use those as a guide for future performance or resource usage [58]. This, although useful, is not of much use when rendering a new dataset never tested, or experiencing environmental changes that never happened before. A more robust solution is to have at disposition an accurate model of the system; having a model at disposition has also the added advantage that many different scenarios can be tried before running, helping to choose the best parameters for a desired outcome.

As we described in the literature review (see 3.4.5 on page 67), much work has been done on adding system prediction to Grid systems.

While detailed models, particularly if based or adapted to the measured system, give better prediction, black-box models using parametric models (*e.g.* [16]) are particularly interesting to us, in that they do not need detailed information to be useful.

This makes them applicable to a vast range of systems without needing extensive modifications (*e.g.*, to add measurement probes) or extensive modelling work (to create or adapt a model, and to validate it).

A side effect of these models is faster evaluation time as they tend to be simpler, which makes them even more attractive for real-time systems such as ours.

We will explore in this chapter a few models and simulation techniques we implemented in our agent system, as well as use cases where we took advantage of predicted results to improve performances.

8.1.1 Amdahl's Law: Evaluating T_p and T_s

Amdahl's law (see section 2.6 on page 18) offers a simple parametric model of the performance evolution of a distributed system. It is expressed in terms of percentage of instructions, parallelisables and non-parallelisables. If we want to use this model to evaluate our system, we need to determine those percentages. While we cannot easily instrument a working system to have detailed timings of the computations to

determine those percentages, we can instead rely on a set of timings $T(n)$ depending of the number of nodes n, to try to compute a value of T_p, the original computation time that varies with the number of nodes (*i.e.* representing the parallelisable part of the computation).

Expressing the law in terms of timings, and with $T(1)$ the computation time with one node and $T(n)$ the computation time with n nodes, we get:

$$T(n) = \frac{T(1)}{S(n)} \tag{8.1}$$

$$T(n) = T(1) \times \left((1-P) + \frac{P}{n}\right) \tag{8.2}$$

$$T(n) = \frac{P}{n} \times T(1) + (1-P) \times T(1) \tag{8.3}$$

or simply:

$$T(n) = \frac{T_p}{n} + T_s \quad \text{with} \begin{cases} T_p &= T(1) \times P \\ T_s &= T(1) - T_p \end{cases} \tag{8.4}$$

Let's have $T(A)$ and $T(B)$, two rendering times for respectively A and B number of nodes. It follows from equation 8.4 that we can write:

$$T(A) = \frac{T_p}{A} + T_s \tag{8.5}$$

$$T(B) = \frac{T_p}{B} + T_s \tag{8.6}$$

It is then straightforward to express T_s (the non-parallelisable time) as:

$$T_s = T(A) - \frac{T_p}{A} = T(B) - \frac{T_p}{B} \tag{8.7}$$

Rearranging:

$$T(A) - T(B) = -\frac{T_p}{B} + \frac{T_p}{A} = \frac{-A \times T_p + B \times T_p}{A \times B} \tag{8.8}$$

$$= \frac{T_p(B-A)}{A \times B} \tag{8.9}$$

We can then write T_p as:

$$T_p = \frac{(T(A) - T(B)) \times A \times B}{B - A} \tag{8.10}$$

Using equations 8.7 and 8.10, we can determine the T_p and T_s parameters using only two data points.

8.1.2 Non-linear Regression Techniques

Another approach to determine Amdahl's P parameter is to use regression techniques to fit the parameter using experimental data points. We used the Levenberg-Marquardt algorithm [169], a popular algorithm providing an iterative solution to the problem of minimizing a function over a space of parameters. We used equation 8.2 on the preceding page :

$$T(n) = T(1) \times \left((1 - P) + \frac{P}{n} \right)$$

Whence given an inital time $T(1)$, we determine P, the degree of parallelisation of the system. While more CPU-intensive than the preceding algebraic method, this tends to work better with many data points.

8.1.3 Implementation of the Universal Model

Amdahl's law is not the only parametric model we used; we also implemented in our system the universal model described in section 2.2.3.2 on page 19.

We transformed equation 2.7 on page 19 to use timings instead:

$$T(n) = T(1) \times \frac{1 + \alpha(n-1) + \beta n(n-1)}{n} \tag{8.11}$$

We use the implementation of the Levenberg-Marquardt algorithm in our system to compute the parameters α and β. One interesting aspect of this model is that contrary to Amdahl's, we can compute M^* the number of processors giving the maximum performance:

$$M^* = \sqrt{\frac{1 - \alpha}{\beta}} \tag{8.12}$$

8.1.4 Distributed Rendering Techniques: Image-Space and Object-Space

The test cases we will explore are examples of distributed rendering — they are systems designed to visualise datasets, distributing the task of rendering images. Two approaches emerge when discussing distributed rendering (cf 2.3.3 on page 37):

1. image-space distribution
2. object-space distribution

The best approach is ideally a mix of these two techniques, although image-space distribution works extremely well with parallellizable rendering algorithms like ray-tracing and ray-casting, as each ray is independent from the others.

8.2 Distributed Rendering with ParaView

As a first example of using parametrics models to analyse systems, we will use rendering timings from ParaView [1], an application designed to visualize large data sets, and analysis them using Amdahl's law.

ParaView supports a rendering system using an object-based distribution. Table 8.1 on the following page shows some rendering timings [2] using a cluster of 48 Opteron cores running at 2.6 GHz. Each core has 2 GB of memory. The dataset used is the visible human female volume dataset ($512 \times 512 \times 1734$, 909 MB). Timings are in microseconds (μs).

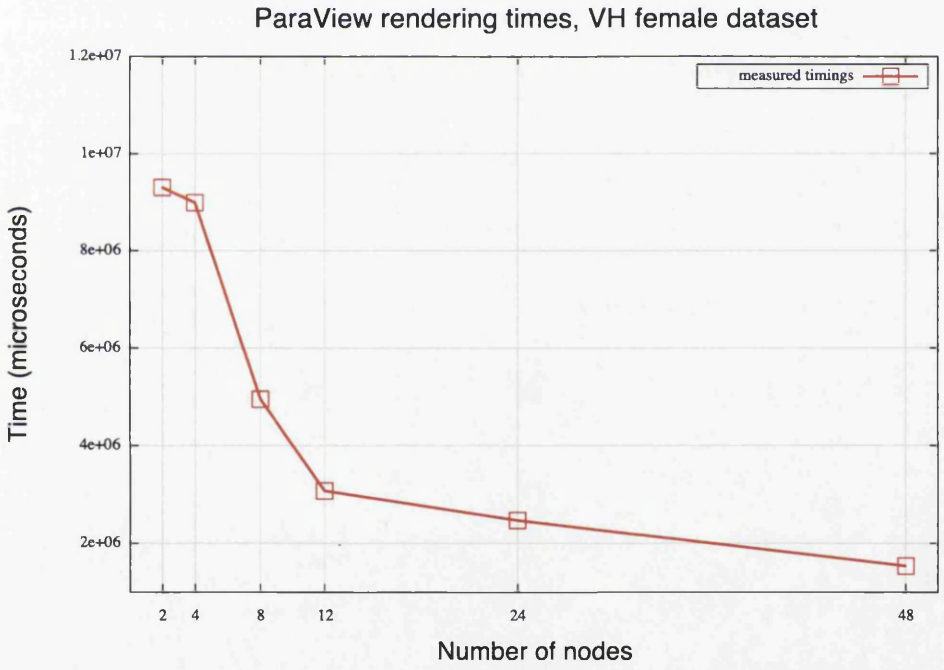


Figure 8.1: *Rendering times of the Visible Human female dataset using ParaView*

Figure 8.1 displays these rendering times on a graph. As we can see, the time is not linearly decreasing, but seems to follow Amdahl's law.

However, it is clear that it does not follow it continuously. In particular there is a definitive change between the behaviour of the system defined up to $n = 4$ and

[1] `http://www.paraview.org`

[2] Timings courtesy of Mark Riding, University of Manchester

Nodes (n)	Render time in μs $T(n)$	Images/s
48	1532199	0.652
24	2467769	0.405
12	3074631	0.325
8	4947570	0.202
4	8986539	0.111
2	9304928	0.107

Table 8.1: *Rendering times of the Visible Human female dataset using ParaView (figure 8.1 on the preceding page shows a graphical representation of these timings)*

after it – the timings suddenly go down much faster than they should according to Amdahl's law.

8.2.1 Experimental Validation

If we take the two first values available to us, at $n = \{2, 4\}$, and apply the equations 8.7 on page 152 and 8.10 on page 152:

$$T_p = \frac{(T(A) - T(B)) \times A \times B}{B - A} \tag{8.13}$$

$$= \frac{(9304928 - 8986539) \times 2 \times 4}{4 - 2} \tag{8.14}$$

$$= 318389 \times 4 = 1273556\ \mu s \tag{8.15}$$

$$T_s = T(A) - \frac{T_p}{A} = T(B) - \frac{T_p}{B} \tag{8.16}$$

$$= 9304928 - \frac{1273556}{2} = 8986539 - \frac{1273556}{4} \tag{8.17}$$

$$= 8668150\ \mu s \tag{8.18}$$

We can now compute $T(8)$ using these values of T_p and T_s, following equation 8.4 on page 152:

$$T(n) = \frac{T_p}{n} + T_s \tag{8.19}$$

$$T(8) = \frac{1273556}{8} + 8668150 \tag{8.20}$$

$$\approx 8827344\ \mu s \tag{8.21}$$

Yet, the actual measured value for $T(8)$ is 4947570 μs – a 78% difference. And it keeps getting worse – reusing those same values we compute $T(12)$ (our next measured value), which gives us $T(12) \approx 8774279$ μs – a 185% difference from the measured time of 3074631 μs.

A possible explanation is that the system had a bottleneck under $n = 4$, and somewhere in the range $4 < n \leq 8$ that bottleneck was passed and the system started to change regime and to scale.

As we are working with an object-space renderer, it is likely that the renderer was not able to cope well with a big dataset on a few nodes, but once the data was split into small enough subsets the system was able to scale.

As our current estimation does not cover this change of regime, we compute $T(n)_{(4,8)}$, the simulation using T_p and T_s for $n = \{4, 8\}$:

$$
\begin{aligned}
T_p &= \frac{(8986539 - 4947570) \times 4 \times 8}{8 - 4} \\
&= 4038969 \times 8 = 32311752 \\
T_s &= T(A) - \frac{T_p}{A} \\
&= 8986539 - \frac{32311752}{4} = 8986539 - 8077938 = 908601
\end{aligned}
$$

n	measured $T(n)$	simulated $T(n)_{(4,8)}$	error (%)
2	9304928	17064477	83 %
4	8986539	8986539	0
8	4947570	4947570	0
12	3074631	3601247	17.1 %
24	2467769	2254924	9.4 %
48	1532199	1581762	3.2 %

Table 8.2: *Error difference between the simulation and the real measures*

We can see from table 8.2 that even if $T(n)_{(4,8)}$ has a much more reasonable error in average than $T(n)_{(2,4)}$ (obviously, not counting the interval $2 \leq n < 4$), it is not perfect. Figure 8.2 on the following page shows in addition to the measured values the simulation $T(n)_{(2,4)}$ and $T(n)_{(4,8)}$.

8.2.2 Discussion

By construction those different $T(n)$ are exact in their prediction between each pair, but fall flat on any regime change. By computing the simulated $T(n)$ incrementally on a running system (instead of computing a small number of them, after the test,

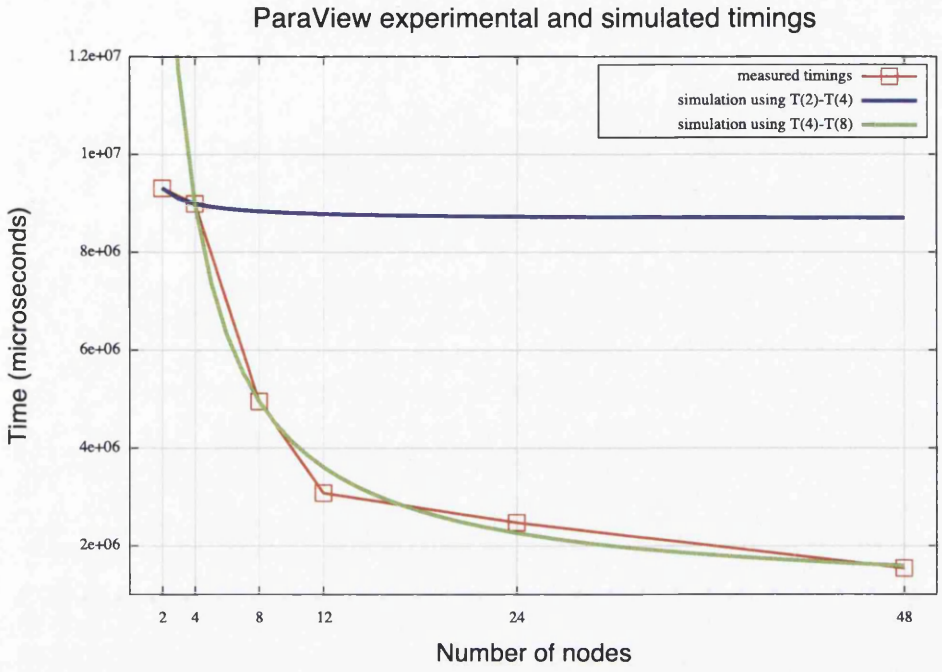


Figure 8.2: *Experimental and simulated rendering times of the Visible Human female dataset using ParaView*

like we did in this example) we can alleviate this problem, as we will then be able to automatically switch to the $T(n)$ with the lowest error with respect to the currently measured performance, and use it to predict the future performance.

If we experience a regime change, we will then automatically switch to a $T(n)$ modelling the regime change. Alternatively we could just change $T(n)$ if the error goes above a specified threshold, if we wanted to avoid jitter.

Using Amdahl's law to predict the system performance is therefore quite practical, even with as little information as we had here (just the total rendering time). But it is necessary to refine the equations while increasing the number of nodes, to take into account any change of regime (bottlenecks).

8.3 A Modelling Agent

Figure 8.3 on the next page shows the general architecture used for this agent. We want to generate automatically a simulation of the running system, in order to use that simulation to predict future performances. We consider here only a generic rendering agent. That agent (in addition to its normal rendering duties) regularly sends to a *Blackboard* agent (basically, a component dedicated to storing information) the rendering times and the number of nodes used in the rendering process. We then

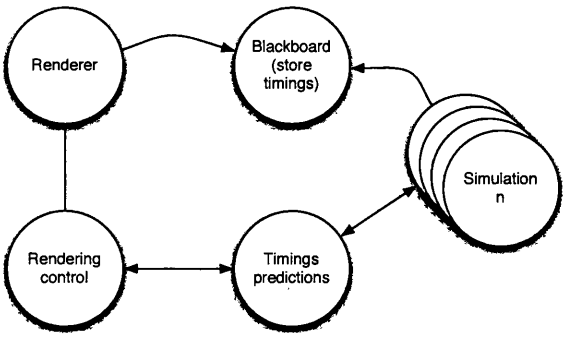


Figure 8.3: *Automatic modelling agent*

have one or more simulation agents using those timings to compute a simulation, and a timings predictions agent that keeps check of those simulations and elect the simulation which is the more accurate at an instant t. A control agent then exploits the simulation data to take a decision and possibly modify the rendering process (*e.g.* asking the system for more rendering agents or conversely freeing some agents).

Figure 8.4 on the following page shows the result of the automatic simulation architecture using the data from the previous section (table 8.1 on page 155). As we can see, the simulation isn't very smooth as we have only 6 control points and we are thus using a linear interpolation (red line joining the points on figure 8.4 on the following page) to check the error difference between the measured values and our simulation. Note that this graph shows the final simulation curve, taking into account all the measured values; but obviously the interest of the simulation is to be able to predict future behaviour, *i.e.* while having only a partial number of measures.

8.4 Bottleneck Identification

Simulation models are subject to inaccuracy, and may deviate from real-life results. But when this happens, there are two possibilities: either the model is wrong, or the system is behaving inconsistently. As such, simulation models can be used to identify potential problems in a running system. We describe in this section a particular performance problem that our distributed visualisation system was subject to, how it was identified and solved.

Our distributed visualisation system, contrary to ParaView, is based on an image-space distribution method. The system is running on a cluster of Xeon 2.66 GHz with 2GB of RAM. The dataset visualised is the visible human dataset, normalized, $512 \times 330 \times 1877$ short int, 600 MB.

Figure 8.5 on page 160 shows how a particular image is decomposed into tiles that

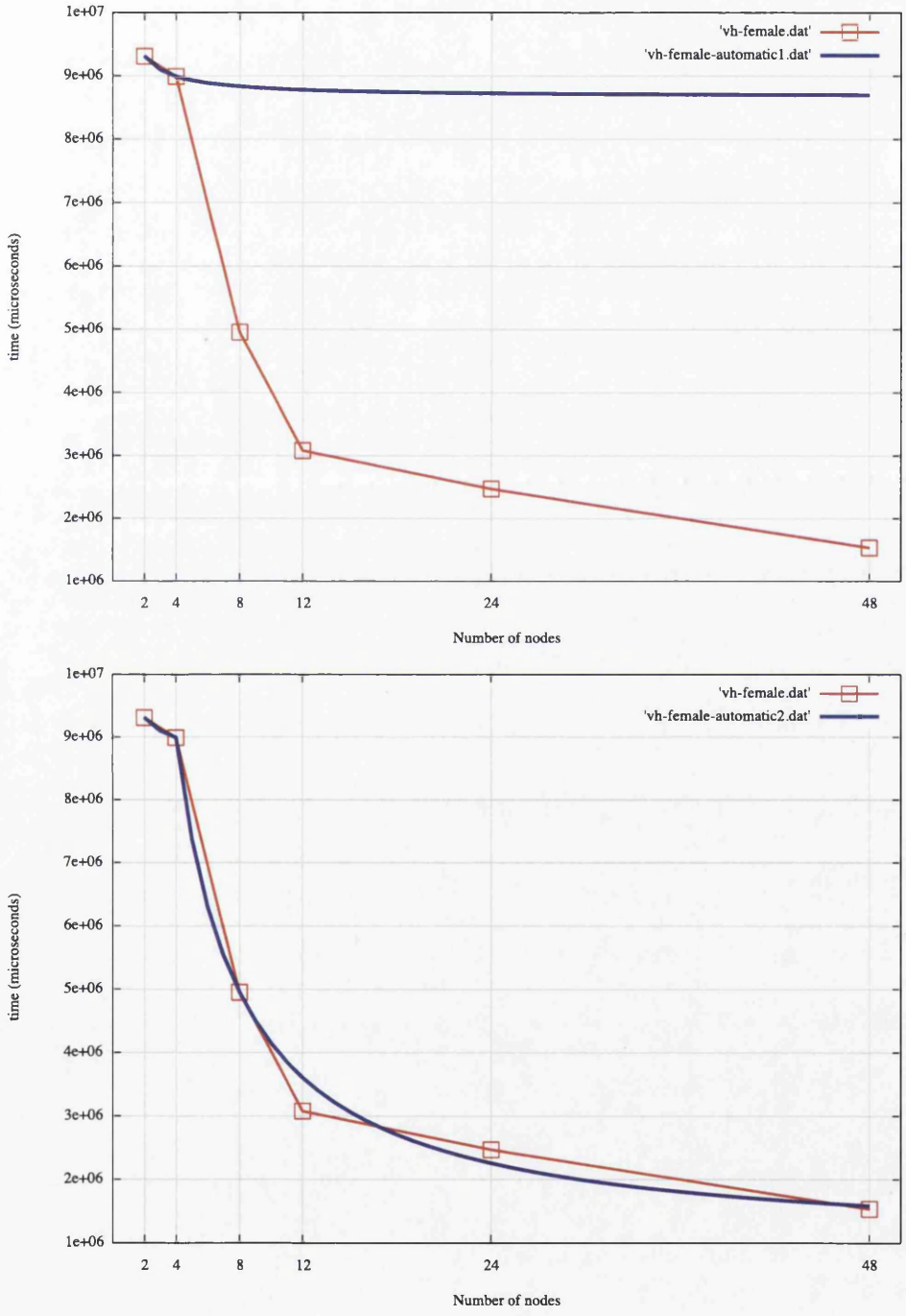


Figure 8.4: *VH female, evolution of the simulated timings*

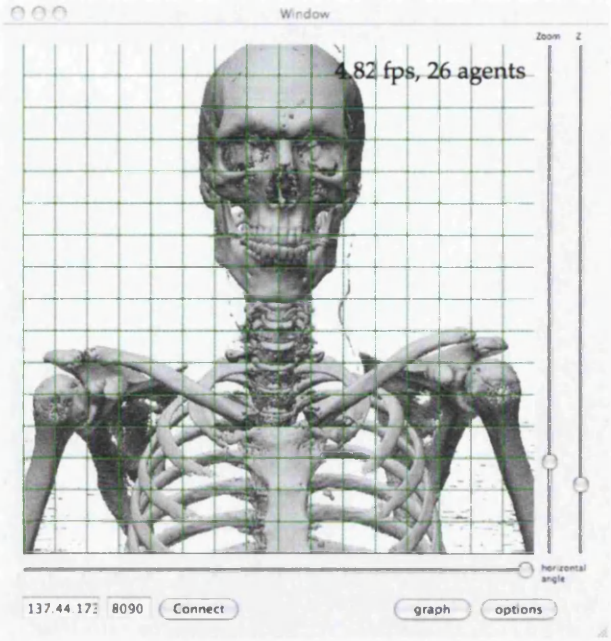


Figure 8.5: *VH dataset, image-based rendering*

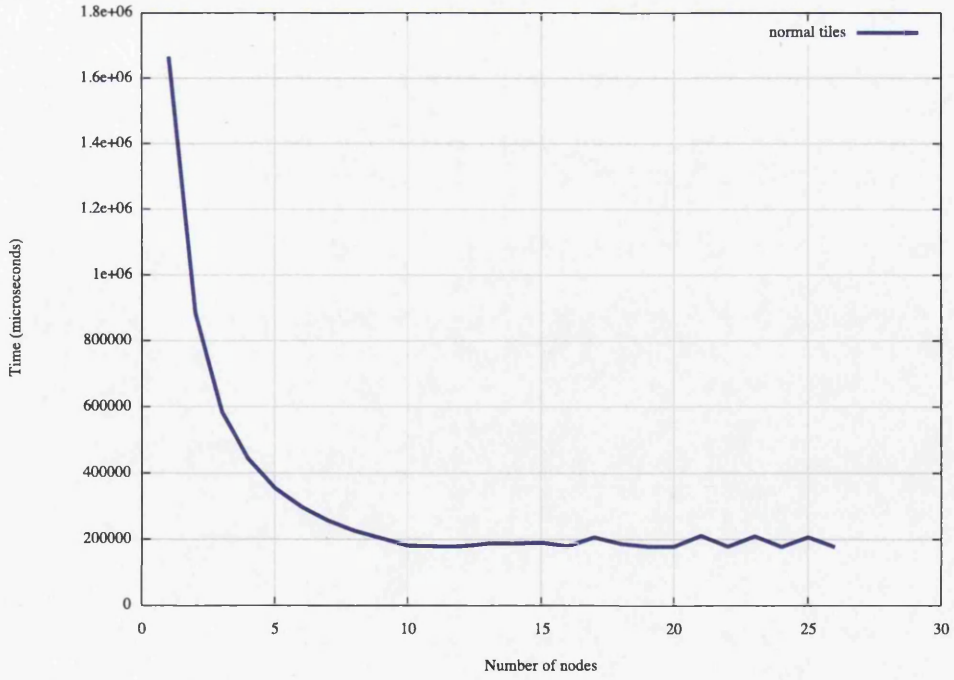


Figure 8.6: *VH dataset, timings with normal tiles distribution*

can be independently rendered (here we have a 512×512 image decomposed in 256 tiles of 32×32 pixels). Table 8.3 on the following page shows the values measured. Figure 8.6 on the previous page shows how performance scales with the number of nodes.

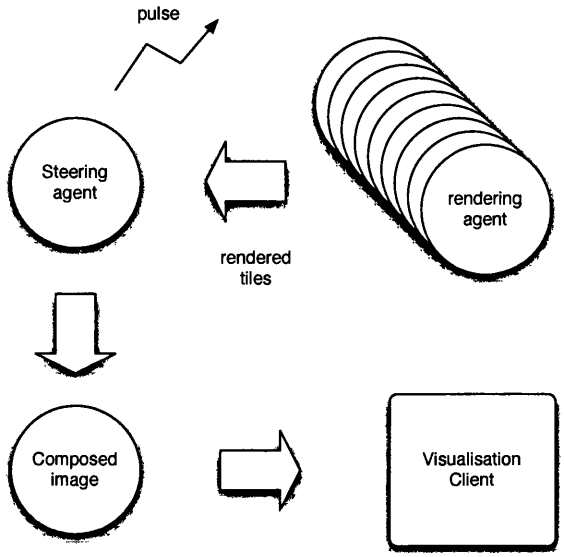


Figure 8.7: *Distributed Rendering system*

Figure 8.7 details how the system worked. We had a steering agent that regularly sent a pulse (a broadcast UDP packet) indicating the need for a new image.

The pulse packet contains the viewpoint and a few other parameters (*e.g.*, final image size). The rendering agents are distributed on the cluster and are listening for pulse packets. When started, they are registered to the steering agent, which can if needed use them for a rendering (*i.e.* it asks for a tile repartitioning by sending a broadcast request; registered agents then connect to the steering agent to receive the list of tiles they are responsible for). When receiving a pulse, the rendering agent uses the parameters to generate the tiles it owns, and sends the result to the steering agent. A complete image is then recomposed from the tiles and sent to the visualisation client.

8.4.1 Clustered Tiles

As we can see on figure 8.6 on the previous page, there is a regime change – a bottleneck – around $n = 10$. We can see this bottleneck more easily by transforming the timings into images/second (Figure 8.8 on page 163).

We discovered experimentally that this bottleneck is related to the number of tiles treated by the system (specifically, the compression/decompression of the tiles and

nodes (n)	time (μs)	images/s	time (μs)	images/s
1	1664643	0.6	1621227	0.62
2	882730	1.13	861551	1.16
3	583940	1.71	567423	1.76
4	442395	2.26	427608	2.34
5	354547	2.82	348259	2.87
6	297453	3.36	287304	3.48
7	255979	3.91	250975	3.98
8	224873	4.45	216243	4.62
9	202905	4.93	197868	5.05
10	180405	5.54	176645	5.66
11	178596	5.6	163495	6.12
12	178365	5.61	152181	6.57
13	186655	5.36	138043	7.24
14	186533	5.36	132693	7.54
15	189380	5.28	126238	7.92
16	178840	5.59	117164	8.54
17	205695	4.86	113759	8.79
18	186010	5.38	107810	9.28
19	177617	5.63	101374	9.86
20	177403	5.64	98485	10.15
21	209965	4.76	96506	10.36
22	177585	5.63	91660	10.91
23	208658	4.79	90173	11.09
24	176582	5.66	85224	11.73
25	205889	4.86	84153	11.88
26	176232	5.67	79512	12.58

Table 8.3: *Performance values for the VH dataset. First two columns (after the nodes column) with a normal tiles division, last two columns with a clustered tiles division.*

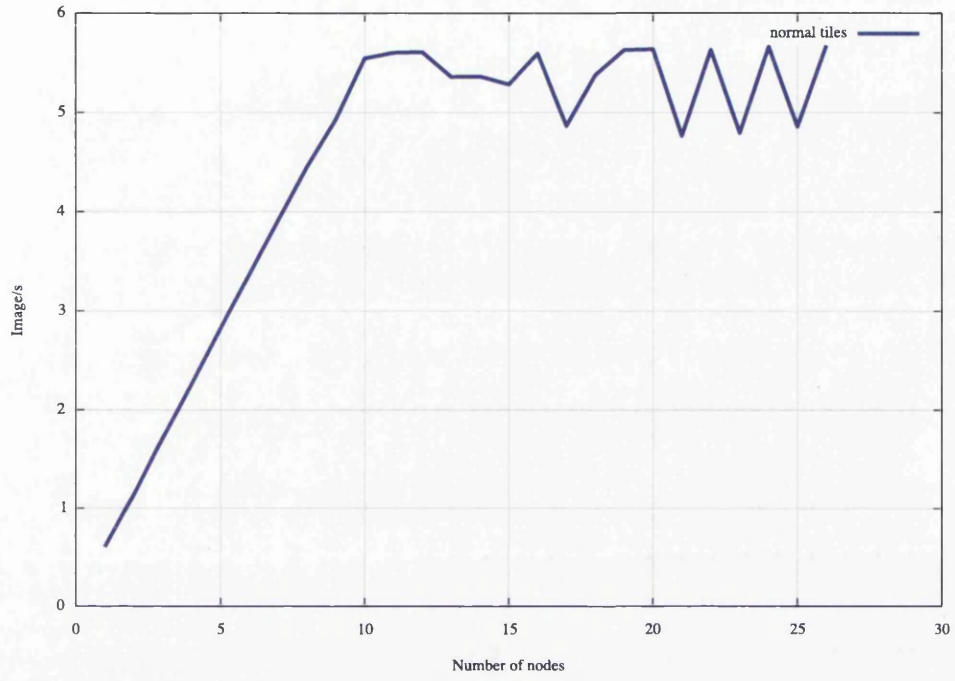


Figure 8.8: *VH dataset, images/s with normal tiles distribution*

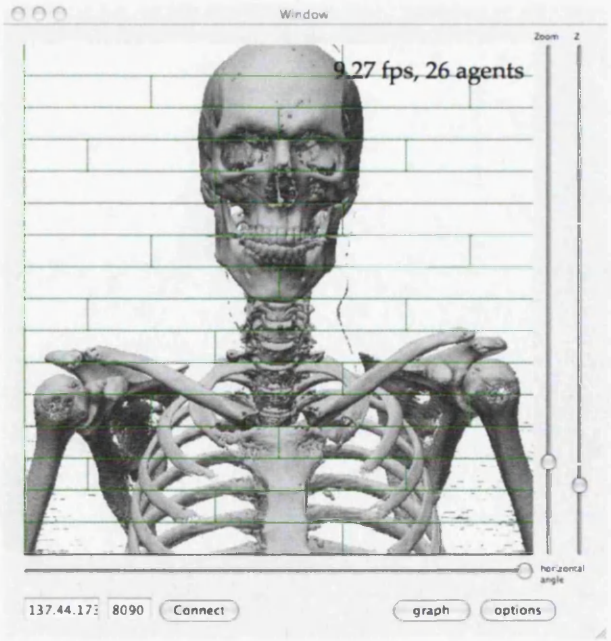


Figure 8.9: *VH dataset, image-based rendering, clustered tiles*

compositing). We chose then to implement a tiles clustering mechanism in order to reduce the number of tiles. Figure 8.9 on the previous page shows an example of such clustered tiles (to compare to figure 8.5 on page 160).

As we can see on figure 8.10, the system scales much better past $n = 10$ and achieves a much higher framerate using the clustered tiles mechanism [3].

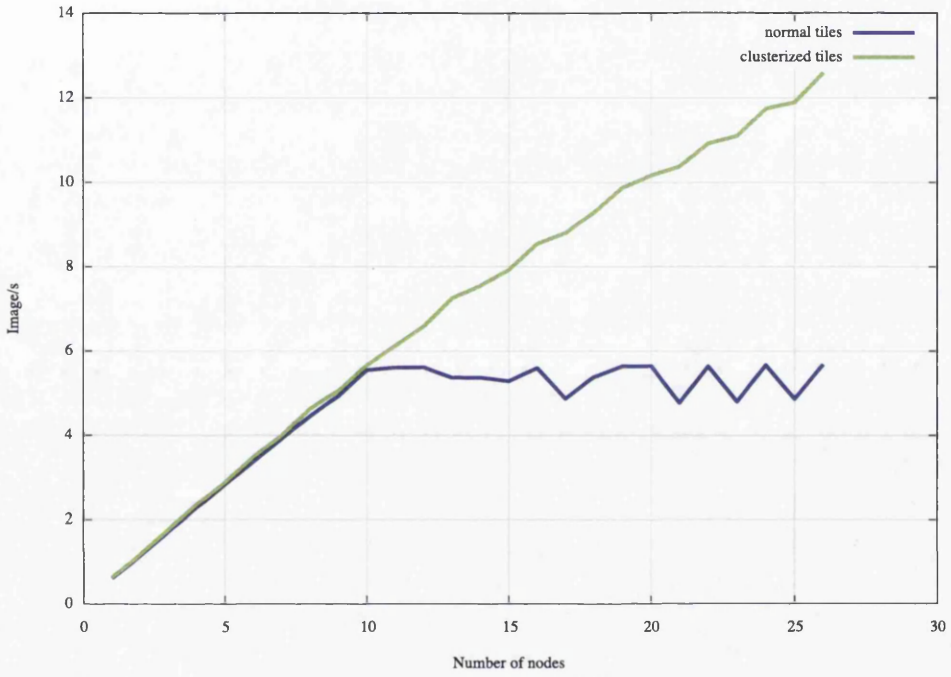


Figure 8.10: *VH dataset, timing results and frames per second*

8.4.2 Using Amdahl's Model Agent in the Rendering Pipeline

As with the ParaView test case, we introduce a set of agents that use Amdahl's law to simulate a running system and make predictions on future performance. Being our own system, those agents integrate directly with the rest of the agents composing the distributed rendering pipeline and we can use the simulation results in real-time.

8.5 Performance Models: Experimental Results

We introduced in the preceding sections different models used for simulation. We added to our system agents implementing those simulation models, in particular

[3] This test case dealt with a previous architecture of our rendering pipeline; the current architecture achieves better results even without clustering.

an agent applying Amdahl's law (figure 8.13 on the following page) to compute parameters T_p and T_s (see section 8.1.1 on page 151 for more details), another agent also using Amdahl's law but using non-linear regression and existing performance data to compute P (figure 8.14 on page 167, and finally an agent implementing the universal model, also using non-linear regression to determine the model's parameters (figure 8.15 on page 167).

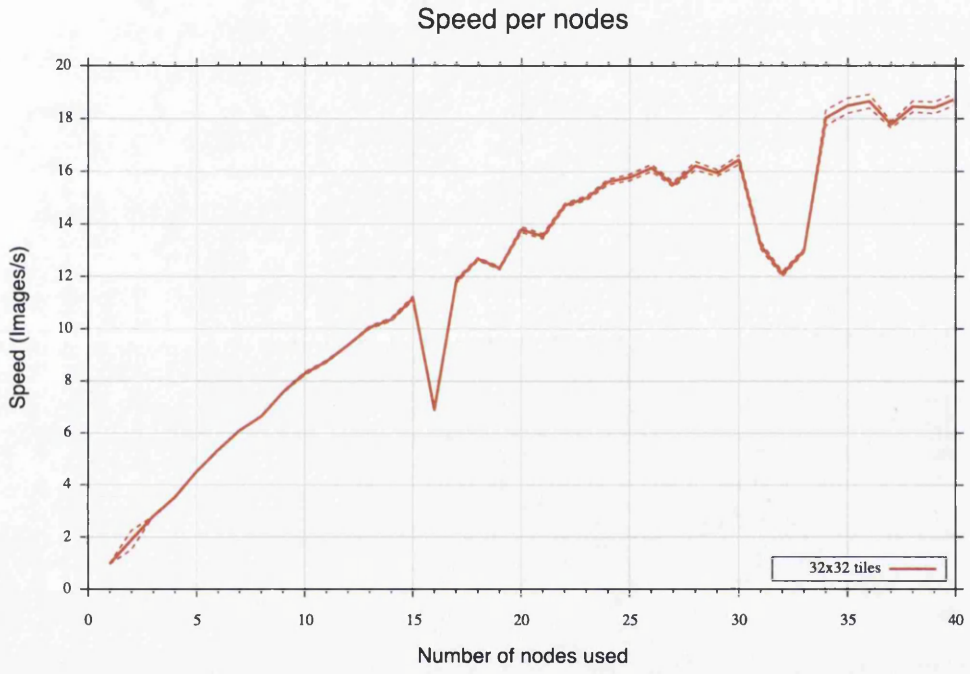


Figure 8.11: *Experimental results (dotted line shows the standard deviation)*

Figure 8.11 shows the experimental results obtained by rendering an image of the visual human dataset with our system (rendered image shown on Figure 8.12 on the next page).

We can see that the system seems to scale up roughly following an Amdahl's progression, with the exception of two speed bumps, when using 16 and 32 nodes for the rendering. Using the "Amdahl agents" (Figures 8.13 on the following page and 8.14 on page 167) we can see how the performances match the prediction closely, excluding the speed bumps at nodes 16 and 32. A difference between the two agents is that the agent using the algebraic method to solve the parameters returns the curve passing by at least two points and having the lowest deviation; the curve computed at node 12 is thus kept as it is the one with the least deviation overall. The second agent on the other hand uses non-linear regression and determine a curve minimizing the distance with all the points; there are more separate curves, but they tend to be closer.

Figure 8.15 on page 167 shows the results obtained by the simulation agent using

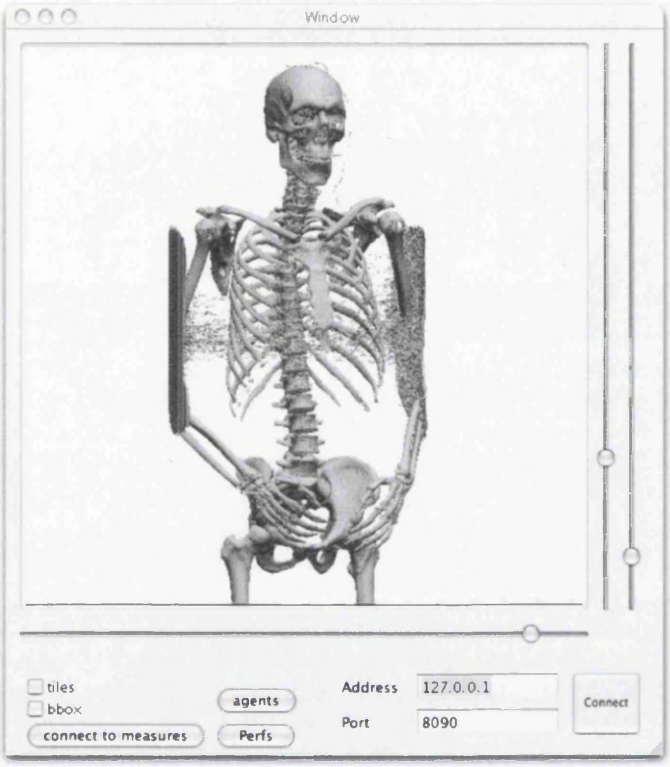


Figure 8.12: *Visual Human dataset rendering*

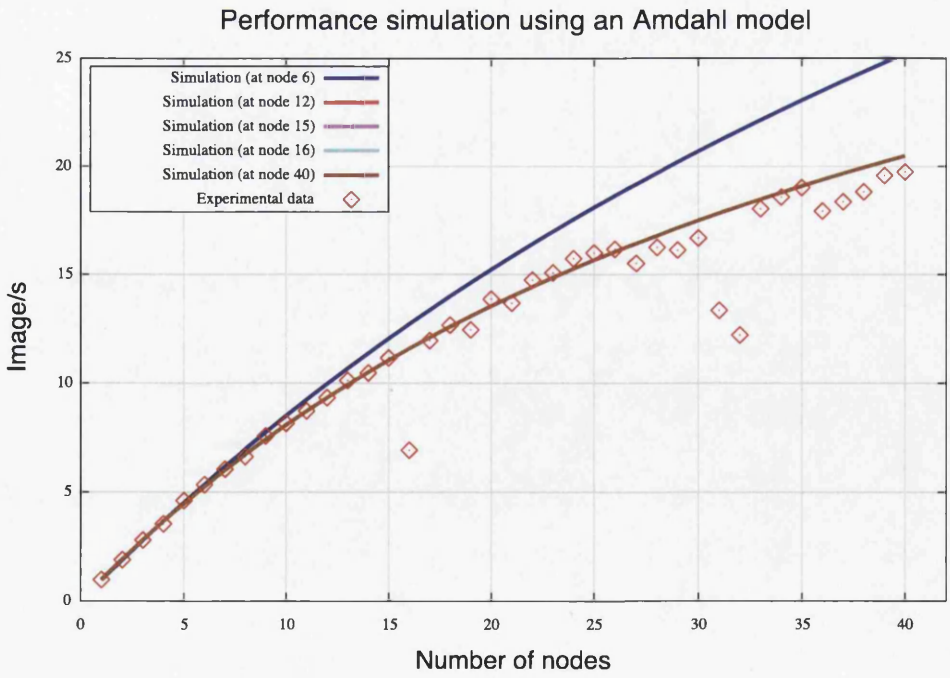


Figure 8.13: *Simulation using Amdahl's law*

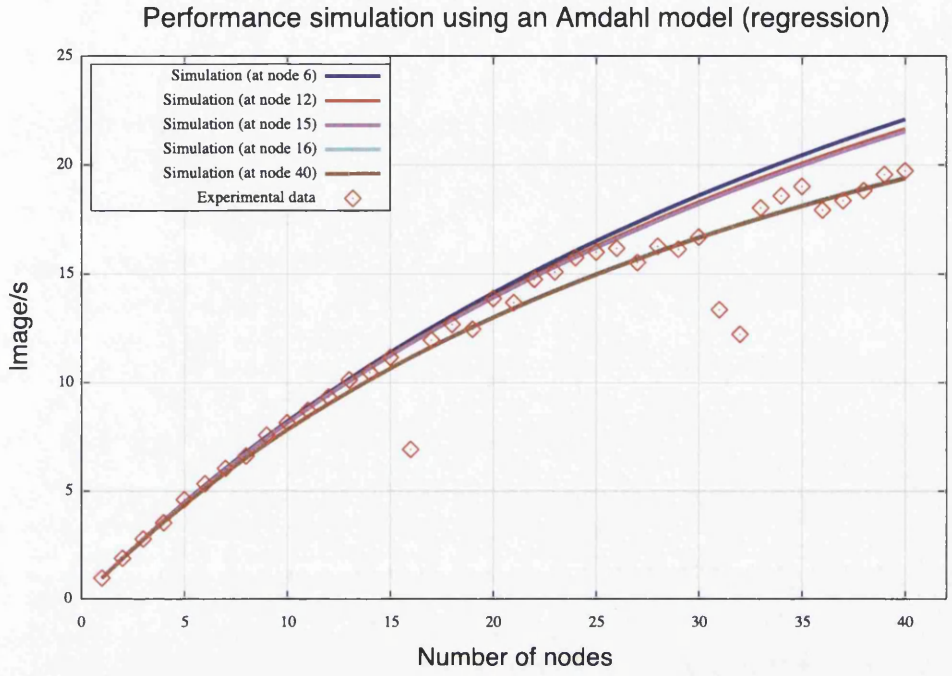


Figure 8.14: *Simulation using Amdahl's law (nonlinear regression)*

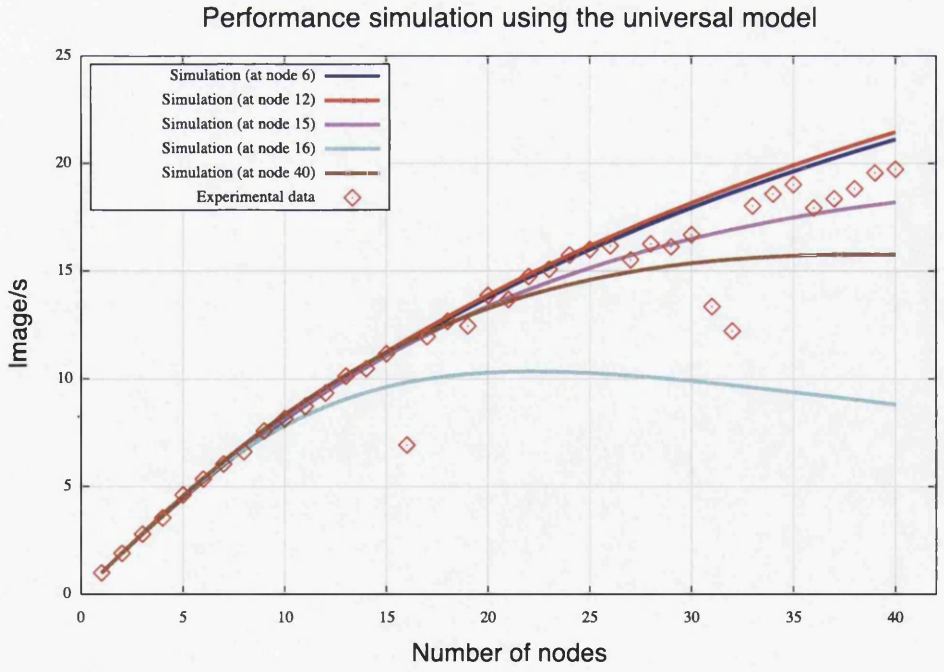


Figure 8.15: *Simulation using the universal model (nonlinear regression)*

the universal model. As we can see, this is a much more pessimistic model and in this particular set of experimental data its prediction value is not particularly great, as the system scales very well. It is nonetheless an interesting model as it can help foreseeing performance limits, by computing its maximum, allocating it and refitting the model to the newly gathered data.

8.6 Arbitrage Agent

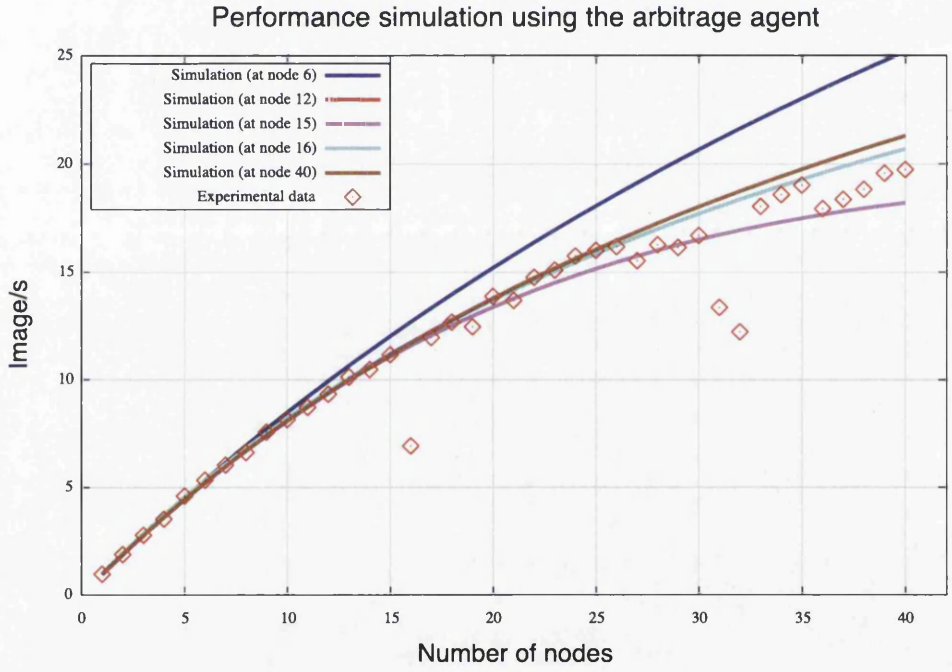


Figure 8.16: *Simulation using the arbitrage agent*

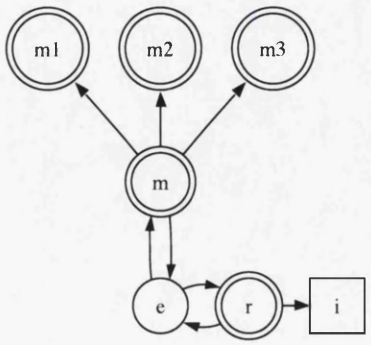


Figure 8.17: *Arbitrage agent in a graphic pipeline*

Figure 8.16 shows the prediction returned by an *arbitrage* agent; this agent does not

do any simulation itself, but simply uses the existing simulation agents to pick the simulation that is the most accurate according to existing results. As we can see, it uses at first the universal model (blue line) as it is more accurate, in part due to the speed bumps, and later on (from 16 nodes) settles on the amdahl model as the system continues to scale up.

Figure 8.17 on the preceding page shows the general architecture of the arbitrage agent. Agent m is the arbitrage agent, using three different models m_1, m_2, m_3. The pipeline (composed of an environment e, a rendering agent r generating the image i) interacts with the arbitrage agent m to improve its performance.

8.7 Rendering Complexity and Load Imbalance

A useful measure of parallel performance is the serial fraction of a parallel system, introduced by Karp and Flatt in [142]:

$$f = \frac{1/S - 1/P}{1 - 1/P}$$

where S is the system speedup, calculated using $S = \frac{T(1)}{T(P)}$, P the number of processors and $T(n)$ the time associated with the number of processors used.

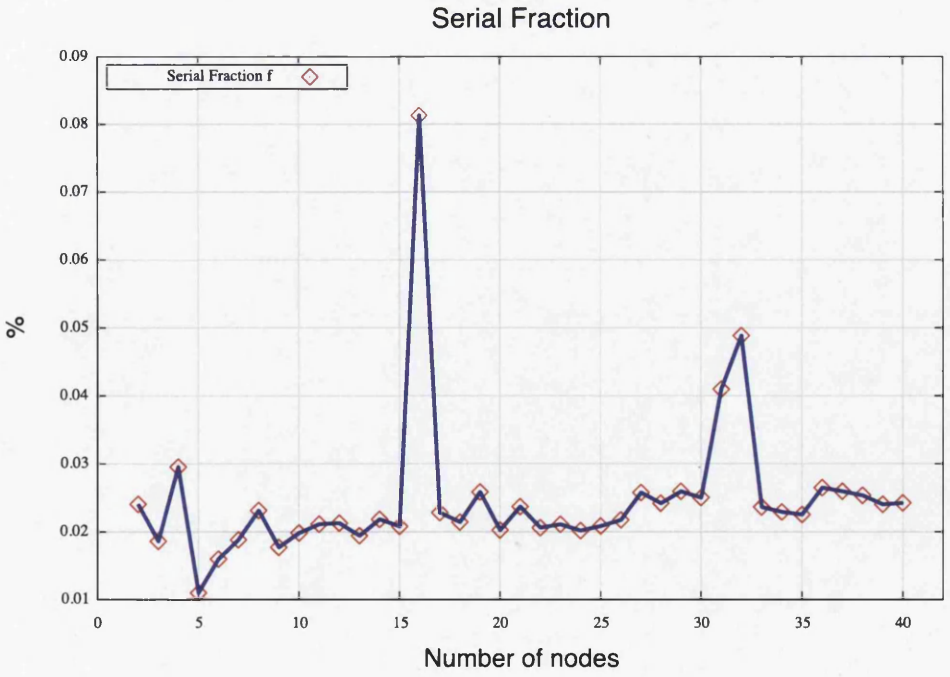


Figure 8.18: *Serial Fraction f*

This metric is interesting for us as it reflects the load imbalance of a system: a perfectly distributed system should have a constant serial fraction. It is easier to identify variations of something ideally constant than unexpected variations of a variable.

In our case study, the computed f value is shown on figure 8.18 on the previous page. We can see that f is mostly bounded in the 2-3% range, apart from nodes 16, 31 and 32. This sudden increase in f is the mark of a sudden load imbalance in the system.

We saw that the different models work rather well to identify the general tendency, but are entirely oblivious to the speed bumps found at node 16 and 32. We can in fact easily understand the reason for those speed bumps; with an image-space distributed rendering, the image is split into different tiles that are then rendered by agents. If the rendering cost was uniform across the image, it would not matter which tiles were sent to which agent, but simply how many tiles are sent per agent (providing the tiles are of equal size). Of course, the rendering cost per tile is far from being equal, as Figure 8.19 on the following page explicitly shows, and this causes our problem as we will see.

As the tiles' rendering complexity is not homogenous, it is important for the distribution system to try to equalize the amount of work done by each rendering agents. Tiles are split into "buckets", with every bucket of tiles being treated by one rendering agent. The system can then simply try to homogenise the buckets rather than the tiles.

Figure 8.20 on page 172 and 8.21 on page 173 shows the actual rendering cost per bucket for 15, 16, 17 nodes and 30, 31, 32, 33, 34 nodes. As we can see, while the cost is roughly similar for 15 and 17 agents, there is an important imbalance for 16 agents. Equally, while the cost is comparable for 30 and 34 agents, there is an imbalance for 31, 32 and 33 agents, causing the overall rendering time to be longer than otherwise expected.

8.7.1 Distribution Algorithms

As shown in the previous section, we know that the complexity is not constant; therefore, the way we distribute the tiles among the rendering agents will impact the overall rendering speed.

Distribution algorithms compute the actual sets of tiles $T_\sigma(a)$ per agent a. We describe here a few possible distributions. All the examples use results from a Visible Human dataset rendering, with a 512×512 image and 32×32 pixels tiles, for 10 rendering agents.

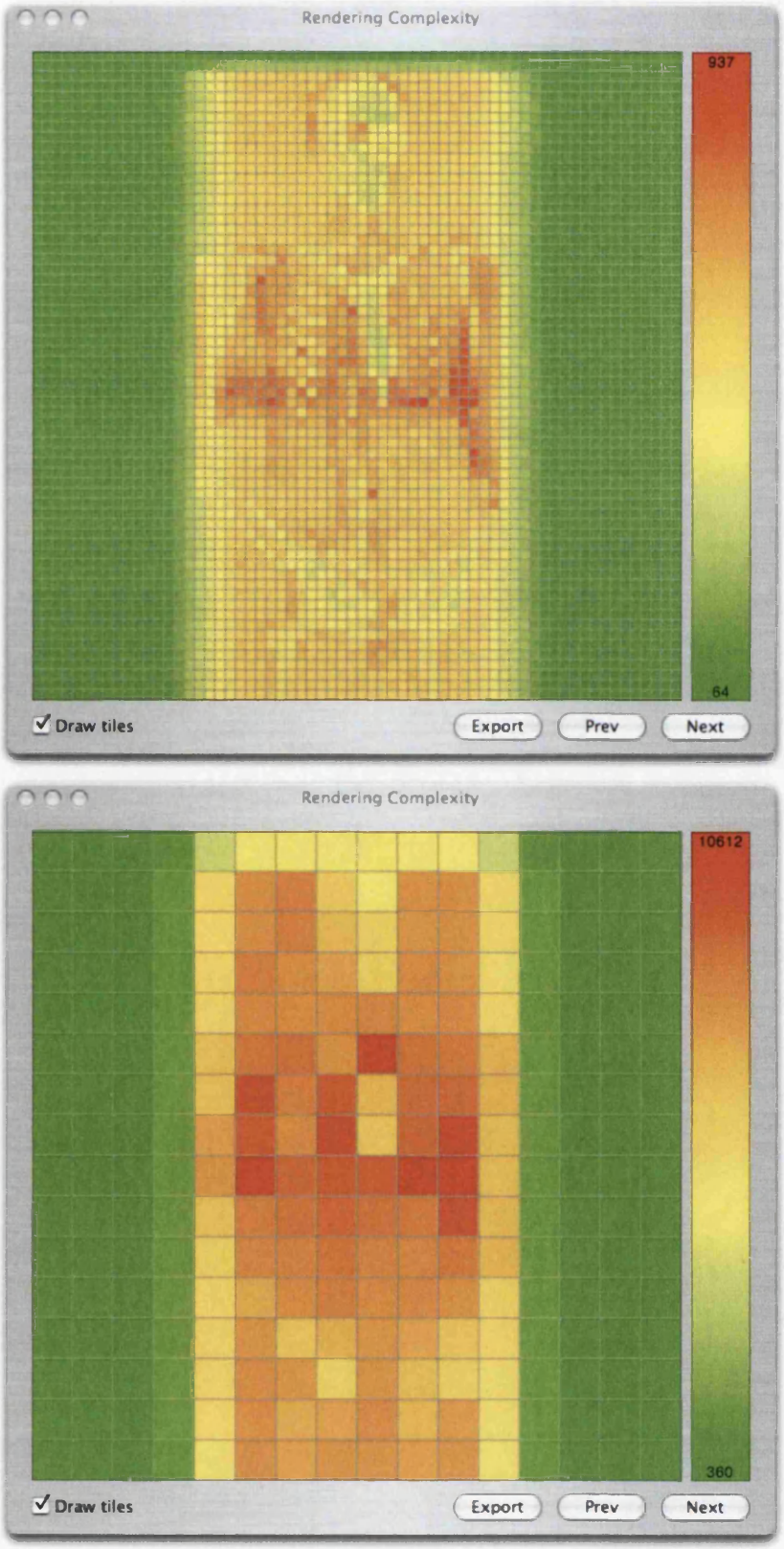


Figure 8.19: *Image complexity (8 × 8 tiles and 32 × 32 tiles)*

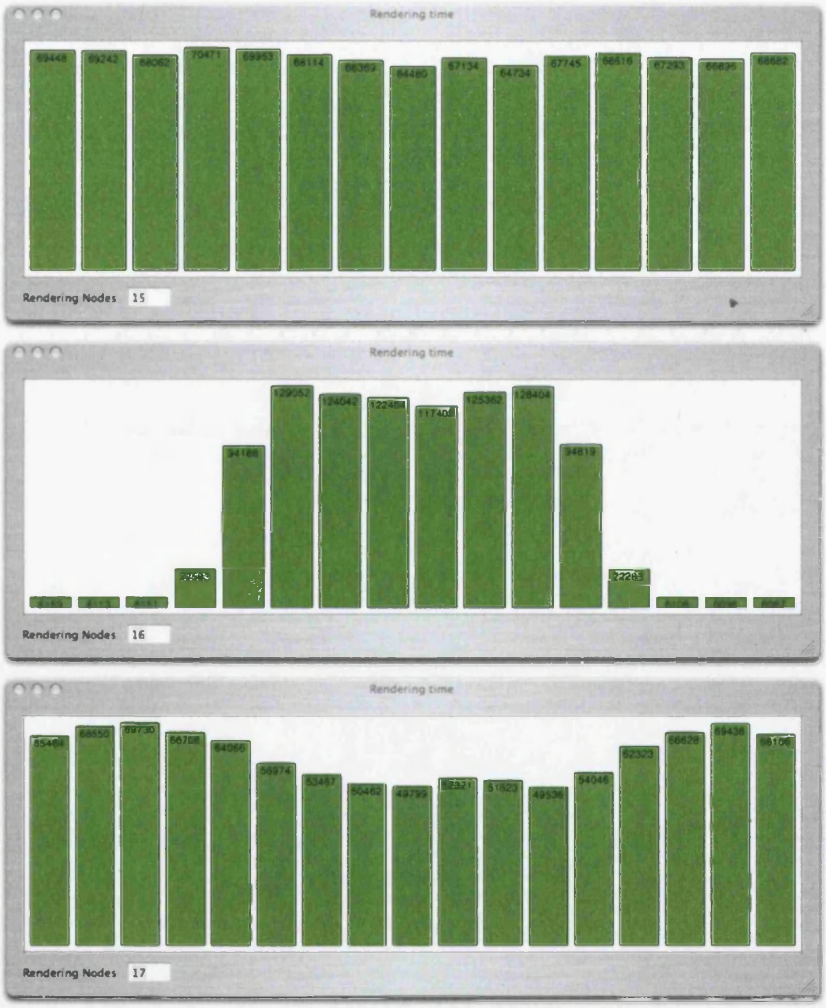


Figure 8.20: *Bucket loads for 15, 16 and 17 nodes, alternate distribution*

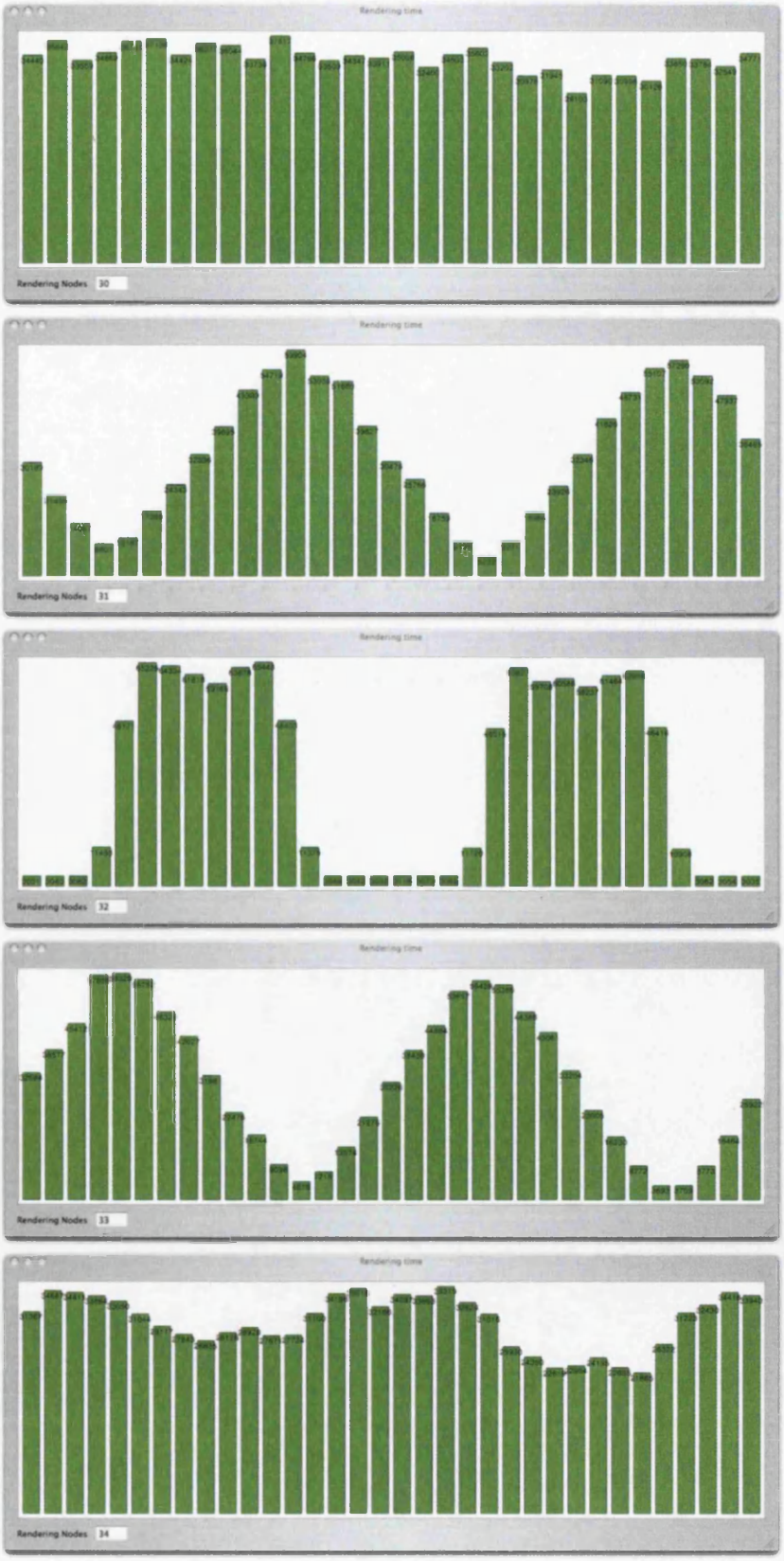
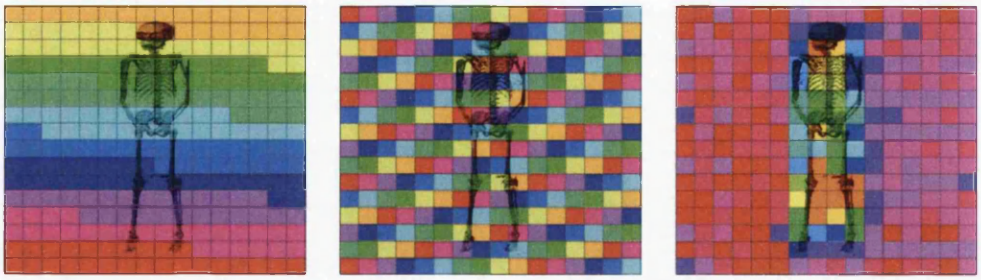


Figure 8.21: *Bucket loads for 30, 31, 32, 33 and 34 nodes, alternate distribution*



(a) Linear Distribution *(b) Alternate Distribution* *(c) Sorted Distribution*

Figure 8.22: *Distribution Algorithms*

8.7.2 Linear Distribution

The simplest distribution consists of dividing the total number of tiles per the number of agents, and associate the same number of tiles per agent. Figure 8.22 (a) shows an example of this distribution (each color corresponding to one agent).

8.7.3 Alternate Distribution

Another simple distribution is close to the linear distribution (each agent receiving the same number of tiles on average), but alternates the agents instead of attributing the tiles linearly. Figure 8.22 (b) shows an example of that distribution. One advantage of this distribution over the linear distribution is that the complexity per agent is much better distributed, which ensures a more constant framerate.

8.7.4 Sorted Distribution

This algorithm uses the rendering complexity information to sort the tiles per agent – agents will not have the same number of tiles on average, but rather, the same complexity on average. Figure 8.22 (c) shows an example of that distribution.

8.7.5 Imbalance

The observed imbalance and its consequences (speed bumps) is a direct consequence of the algorithm we used to split the tiles into buckets,*i.e.* the alternate distribution. While this distribution allows a better balance on average, if the number of rendering agents is a multiple of the number of tiles available in one row, they align themselves as shown on Figure 8.23 on the following page. When the rendered image has a strong imbalance itself, as does the rendered image in our test (see Figure 8.19 on page 171), we end up with a strong bucket imbalance.

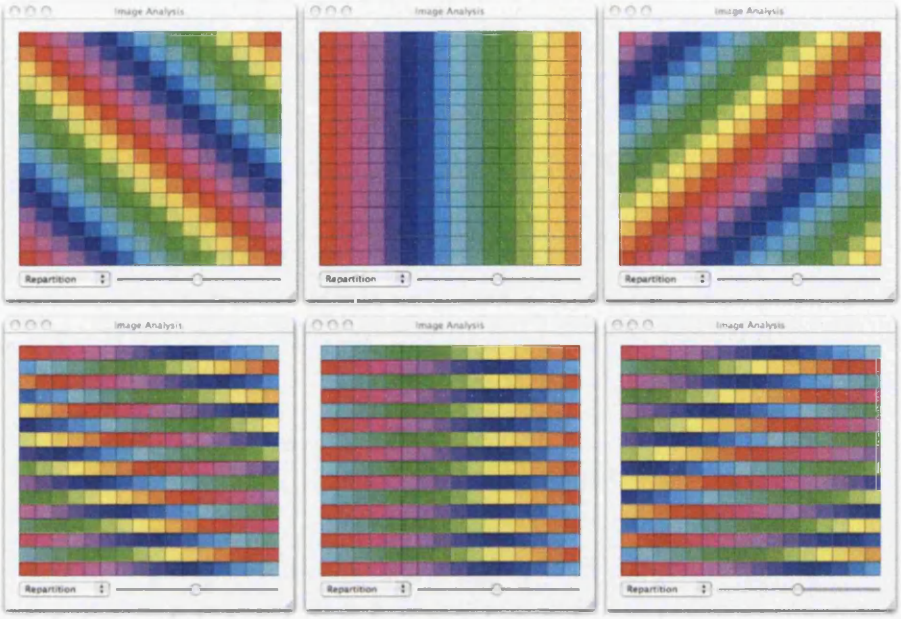


Figure 8.23: *Imbalance caused by the tiles splitting mechanism. Top row respectively shows 15, 16 and 17 nodes used. Bottom row shows the tiles distribution with 30, 32 and 34 nodes. Each color corresponds to the same agent. The central column shows an alignment of agents, producing the buckets imbalance.*

8.8 Use of Simulation in the Graphic Pipeline

In order to solve those runtime imbalances, we implemented a simple feedback agent, similar to the one described in section 6.3 on page 123, which uses the *arbitrage* agent to determine which number of rendering nodes will be optimal in the system.

A second use of simulation technique we used in the system was to simulate the imbalance of the tile buckets for a given number of rendering nodes and a given tiles splitting algorithm; using this information it is easy to determine which particular splitting mechanism works the best. As the splitting only occurs once (when adding or removing nodes) it is a nearly free operation.

Experimentally, we observed that the splitting method that was the most efficient was the alternate distribution method, with the exception of course of the case where the number of agents corresponds to multiple of the number of tiles per row.

Using such a reflective system allows us to always choose the best possible splitting algorithm for a given number of nodes.

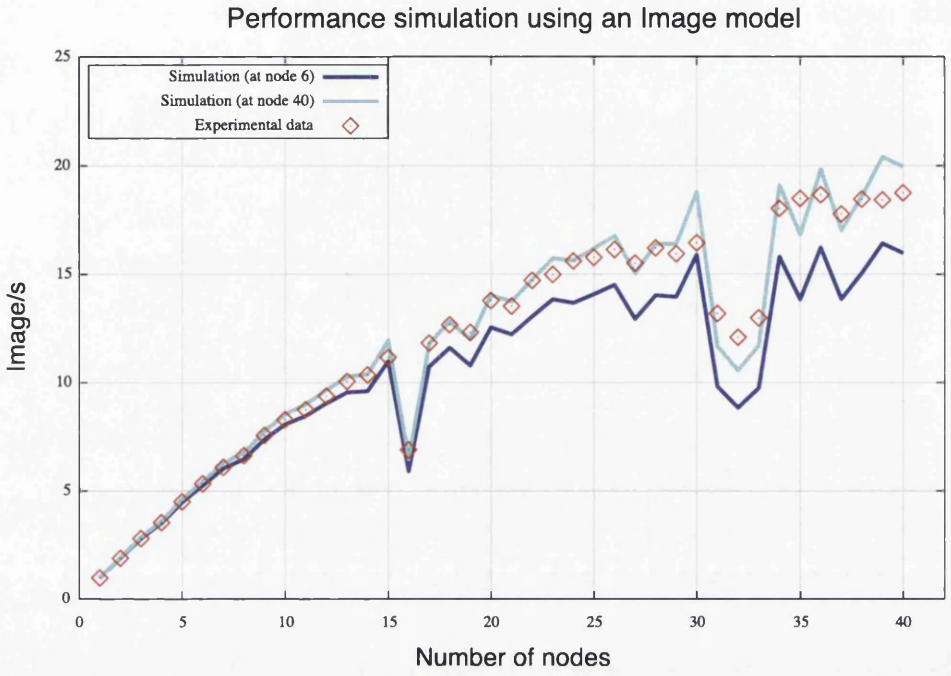


Figure 8.24: *Simulation using the image-space model*

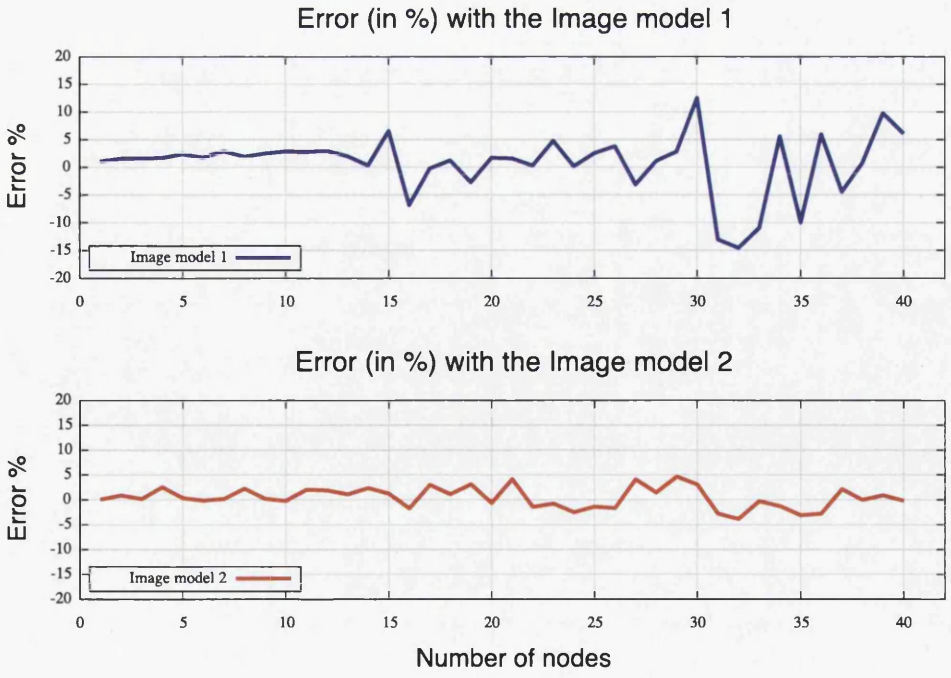


Figure 8.25: *Error (in %) between the image-space models and experimental data*

8.9 A Better Simulation Model for Image-Space Rendering

While the general models introduced in the previous sections are reasonably accurate, they do not consider the imbalance problem we discussed. It is therefore interesting to introduce a model taking advantage of the rendering complexity measures to predict imbalances.

Each tile's rendering time is actually gathered by the distributed graphic pipeline; it is therefore a nearly free operation to have those rendering times examined by another agent (the communication time is the only issue here, and translates to a slight delay, which is generally of no big impact as we take advantage of the fact that frames usually do not differ much from each other).

The agent uses those measures and tile-splitting algorithms (*e.g.*, linear or alternate distribution) to compute an ideal rendering time $T_i(n)$. This ideal rendering time only depends on the number of nodes used. On this ideal time, we apply a speedup factor S, in essence following Amdahl's model:

$$T(n) = T_i(n) \times S \tag{8.22}$$

We use the existing data points as they are made available, and do a non-linear regression on those to determine the speedup factor S.

With this factor S applied to the ideal rendering time $T_i(n)$, we can generate a curve showing the imbalance caused by the type of bucket distribution, and still get close to the actual rendering time via the factor S. Because of S, the resulting curve follows Amdahl's model: with a perfect rendering time (*i.e.*, where the imbalance between tiles bucket is virtually nil) the resulting curve would be Amdahl's curve, as the $T_i(n)$ curve would be linear.

Figure 8.24 on the previous page shows the result of this predictive model. As we can see on the figure, this model accurately predicts the performance bumps. Contrary to the preceding models presented in this chapter which only needed external parameters, *i.e.* considering the system as a black box, this one needs more information (the tiles' rendering times).

Figure 8.25 on the preceding page shows the error in percentage of the model compared to the experimental data. The top graph uses the image model as described in the above paragraphs; the bottom one gives the error of an improved version of the image model, which consists of using the Amdahl and Universal models at the same time, but only using their value if the rendering time given by the simple image model is superior; in essence, we remove from the data used to compute the Amdahl and Universal model the outliers, using the image model as a guide to find those outliers. This gives us more accurate Amdahl and Universal values, with the two models giving us a prediction range for future values. The last operation we do is to simply return the mean of the Amdahl's and Universal model (barring

the outliers, which are returned by the simple image model). The final result is an important reduction in the error and prediction swings, as shown on figure 8.25 on page 176.

An important notion we can draw from this experiment is that while we used a type of application modelling, this was in no way a particularly complex model of the distributed visualisation system — we stayed at a very high level, and the model in question can be applied to many different kinds of system easily, as long as the rendering time per tile is accessible; rather than consider tiles rendering time as something application specific, we can easily abstract this model by considering "computational load buckets" instead. In our case those buckets contain tiles, but any parallel application have an equivalent system to distribute the load. Interestingly, though the model is simple, the experimental prediction is particularly accurate.

8.10 Delphe Lab

Delphe Lab is a software we have written, to simulate a distributed rendering pipeline, feeding on timing data from a running system [4].

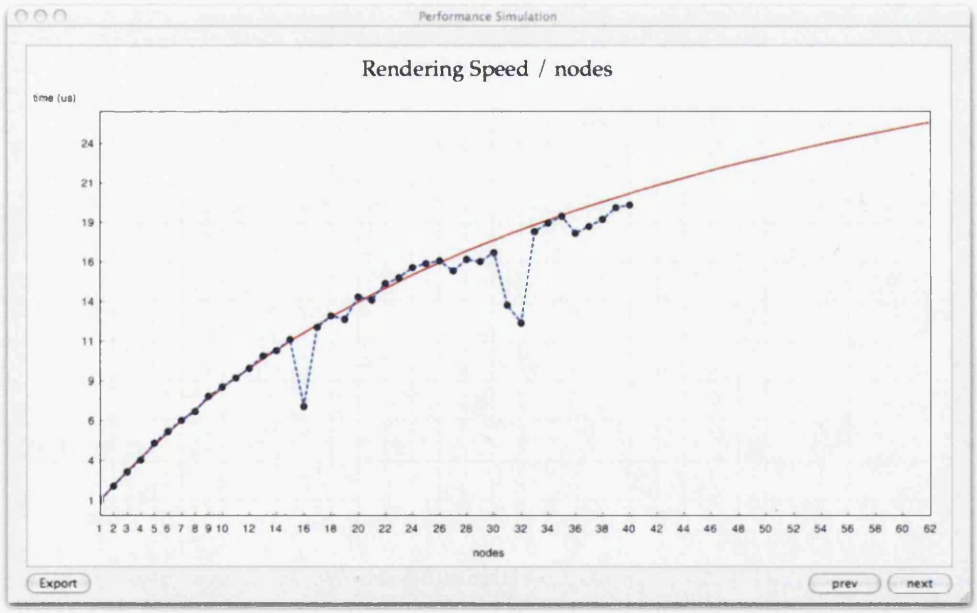


Figure 8.26: *Simulation using Delphe. The dashed line is experimental data collected either at runtime or from a file. The red line shows Amdahl's model applied to the data.*

Figure 8.27 on the following page shows a screenshot of the software running on Mac OS X[5]. The screenshot shows a graph window with various performance

[4] The screenshots of this chapter are from Delphe.

[5] the software also runs on Microsoft Windows and GNU/Linux, using the GNUstep libraries

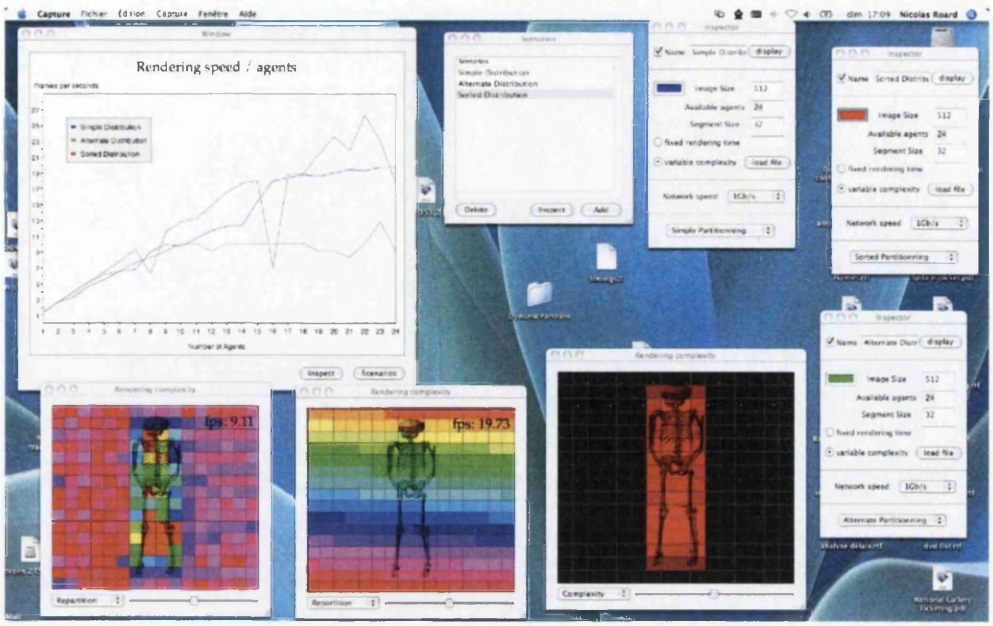


Figure 8.27: *Delphe screenshot, showing a graphical representation of the tile distribution, image complexity and performance simulation using custom scenarios.*

curves given by different scenarios. We are working here on distribution strategies; simulating the different strategies, we can select the best one for the current system capabilities (number of available nodes, etc.).

Delphe lets the user access running graphic pipelines and visualize in real-time the performance of the pipeline, applying in real-time prediction simulations, and technical indicators (*e.g.*, moving average, serial fraction). Using these introspections capabilities, bottlenecks can be more easily identified.

It is also possible to have the pipeline generate performance traces that can be later loaded and analyzed by Delphe.

The knowledge gathered and exposed by Delphe allows further optimisation by users of the pipeline.

Delphe allows to:

- visualise image rendering complexity
- display tile' splitting results using different splitting algorithms
- display tile buckets' rendering loads per node
- experiment with scenarios using simulation models and/or existing data
- models can be written in Objective-C or scripted using StepTalk
- apply to the curves additional technical indicators

8.11 Conclusion

We presented in this chapter two generic performance simulation models for parallel systems, and how we integrated them within our system. Agents implement those models, feeding on runtime performance data, and predict future performances. We saw that even with few experimental data the models were rather accurate and useful.

One important potential of system models is the ability to identify bottlenecks when the experimental data deviate from the model; we described a test case showing a bottleneck we encountered while developing our distributed visualisation system, how the bottleneck was made obvious by the simulation model, and how we solved it.

When analysing experimental data of our final visualisation system with our generic models, we observed that while the models are reasonably accurate, they only provide a general guideline and cannot predict application-dependent performances as accurately; in our example two speed bumps were apparent that the models could not predict. We explained in detail the reason for those speed bumps, and presented one additional model using a small amount of application-dependent information (tiles rendering time) and a speedup constant. This model proved to be much more accurate than the generic models. A final model was then introduced, taking advantage of all the previous models to increase the accuracy.

Other agents in the system (control agents implementing feedback mechanisms in the graphic pipelines) can use these simulations results and the runtime performance data to take autonomous decisions; these possibilities allows for better autonomous behaviours.

Finally, we presented Delphe, a separate program we developed to more easily interact with the system, analyse the experimental data and experiment with future scenarios.

Conclusion

(CHAPTER ...9)

Conclusion

"Heureux qui comme Ulysse
A fait un beau voyage.
Heureux qui comme Ulysse
A vu cent paysages"

— George Brassens

Contents

The main objectives of this thesis as exposed in the introduction were to explore the autonomic computing paradigm, with a focus on visualisation; specifically, we wanted to develop a flexible autonomic visualisation system implementing some of the concepts of visual supercomputing.

9.1 Main Contributions

We first introduced in Chapter 2 a thorough review of parallel computing systems and technologies, defining and exploring the concept of visual supercomputing [35]. In Chapter 3, we presented the problems related to the management of complex systems, different approaches to fault tolerance and reliability, and metrics used for the evaluation of parallel systems. The Grid computing effort and the autonomic computing paradigm were also presented, as well as the work done within the e-Viz project.

Generic Multi-Agent System for Visualisation

Chapter 4 introduced a flexible multi-agent system, implementing mechanisms for communication and cooperation among agents, notably by specifying their relationships and by implementing distribution mechanisms such as Map/Reduce [70] in efficient ways.

Chapter 5 described how it was possible, by using this multi-agent system as a base layer providing common features, to implement a powerful and flexible graphic pipeline. Different types of visualisation clients were created and are described, running on platforms as diverse as mobile devices, desktop computers and the Web. We presented a performance evaluation of the graphic pipeline, reaching interactive framerates with a large dataset (>20 fps with the visible human dataset) using a 512×512 image, without compromising the flexible nature of the pipeline.

A Flexible and Reflective System

The flexibility of our graphic pipeline is one of its most notable characteristics: the ability to change the configuration of a pipeline and the relationship between the components of the pipeline at run time, and by the agents themselves, is one of the key difference between our system and other distributed visualization systems running on clusters.

Chapter 6 showed how this intrinsic runtime flexibility is exploited by the agents, using the reflective capabilities of the pipeline to implement different reflective patterns extending the features of the generic pipeline described in Chapter 5.

In addition to those patterns, the failure protection mechanisms and autonomic features such as auto-configuration, resource discovery and self-healing were presented, also based on the capability of the system to adapt itself at runtime. The adjunction of such autonomic features to a distributed graphic pipeline is of particular interest to us, as most existing autonomic systems do not provide this type of low-latency reaction, and existing cluster visualisation systems are poor in terms of autonomic features.

Chapter 7 described novel reflective patterns, specifically targeted toward visualisation architectures. Those patterns provide reusable components simplifying the creation of complex graphic pipelines, as well as providing generic features to other agents. Notably, new rendering algorithms are able to use those visualisation strategies to gain features transparently.

Performance Evaluation Framework

Finally, Chapter 8 introduced the need for system models, and the simulation and performance gathering capabilities of the system. Different use-cases were then presented showing how those capabilities could be exploited automatically, and a new performance model, combining historical data with a parametric modelling applied to image rendering was described. Delphe, a software analysis tool allowing users to easily explore a running system was then presented. It also allows post-mortem analysis, using performance traces from pipeline runs (more details are available in the appendix).

9.2 Revisiting our Hypothesis

Our principal objective was to explore the creation of an autonomic visualisation system – specifically, how to create a software system tending toward autonomy, while maintaining a certain flexibility, and more importantly, high performance, necessary for the visualisation process.

High Performance

We were able to reach a good framerate (above 20 fps) rendering a 512×512 image of large datasets (between 600 MB and 900 GB for the visible human dataset), distributed on a cluster of machines. In order to do so, we had to develop a custom, performant communication infrastructure, notable by its use of both broadcast and unicast mechanisms and low latency. Another important feature that enabled those performances was the ability to leverage our implementation of a good parallel paradigm, Map/Reduce, allowing us to express distribution problems in high-level terms, while having a very efficient and robust implementation.

Flexibility

The entire system is designed to be flexible, and this is demonstrated most notably by the visualisation and reflective patterns we developped in chapters 6 and 7. A key component of the system flexibility is our initial design decision of basing the system entirely on an agent system; this allowed us to define a robust base we could relay on, as many flexible mechanisms were implemented as intrinsic features, such as the collaboration among components, components allocation, communication and reliability. By leveraging those base features, we developped the innovative approaches presented in chapters 6 and 7.

Autonomicity

Autonomicity is of course an important aspect of our work. The typical features of an autonomic system are *self-configuration, self-optimisation, self-healing* and *self-protection.*

Our system provides some aspects of self-configuration, with pipelines and visualisation stragies automatically allocating necessary components such as rendering agents. Mechanisms such as Map/Reduce and Pool of agents also allowed for such self-configuration of the system.

Self-healing is another important aspect — possibly one of the most important — of an autonomic system. Upon failure, the system should automatically heal itself to a working state. By implementing failure detectors, redundancy of agents, saving state automatically — all features implemented by and with our software agents — and by having as much as possible a REST approach [83], our system can provide such a self-healing capability.

In terms of self-optimisation, our principal efforts as presented in chapter 8 went into developing a performance evaluation framework, with different experimental models (implemented as agents) able to analyse a system. Again, we could leverage the software agents nature of the system by easily implementing an arbitrage agent using the other models and choosing the more accurate. While we could not for practical reasons (the cluster of machines we used to develop our system had been decommissionned by that time) try experimentally a feedback agent using those models, simulation results and previous experimental results of a simpler feedback agent encourage us to think the system proposes a reasonable self-optimisation approach. We also used a software analysis tool we developped, Delphe, for more insight in the bottlenecks of our system and finding solutions to those problems.

We did not consider the problem of self-protection, as our system was not a public one and was of experimental nature. Some aspects of our architecture would limit security problems (such as a clear limit between the system and the visualisation inputs from the users and results), and we believe that implementing a self-protection mechanism and a security policy using agents would be a good solution, as specific security concerns could be encapsulated among different agents and/or in the communication framework.

It is also interesting to look back on the *fallacies of distributed computing* table (Table 3.1 on page 54) we introduced in Chaper 3. Those fallacies represent a list of the classic problems a distributed system can encounter, and provide us with a good comparison tool, as autonomic systems should be able to automatically work around the problems described.

Listing those problems, we answer several of them, notably:

- *Unreliability of the network* We automatically save state, provide redundancy of

agents, and plan for failure, using redundancy in pool of agents as well as a strong REST [83] approach.

- *Latency is zero* While we can only be as reactive as the underlying network, we took a special care in providing fast communication and synchronisation systems, in order to reduce the average latency, resulting in good performances for visualising volume datasets. A low-latency communication architecture is a pre-requisite for a visual supercomputing system.
- *Topology doesn't change* Our reflective system allows us to quickly react to modifications of the network topology, adapting to changing conditions.
- *The network is homogeneous* The communication protocols we implemented are simple and easy to port to different architectures and languages. As an example, the system, as it is, is composed of code programmed in C, C++, Objective-C, Java, Smalltalk and Python. We can integrate in the system different CPU architectures and different type of machines (*e.g.*, mobile devices, desktop, clusters) easily, thanks to the openness of our communication protocols.

Our system thus provide an answer to those common reliability and adaptability problems.

In summary, we presented a case study of a complete visualisation system, allowing autonomic behaviours, permitting distributed visualisation and real-time rendering of complex volume datasets such as the visible human dataset, while keeping good runtime performances and flexibility.

9.3 Further Work

Continuing the previous section's comparison we do not address those particular fallacies:

- *Bandwidth is infinite*
- *The network is secure*
- *There is one administrator*
- *Transport cost is zero*

Most are not critical ("infinite bandwidth", more administrators) for our current needs; but working on enabling a greater security (which would also partially correspond to the *self-protecting* aspect of autonomic computing) would be an interesting challenge; while security is a process that needs to be taken into account from the design phase, we believe the reflective nature of the system makes it possible to implement in our system.

An interesting aspect introduced in [38] is the notion of a reflective user interface, where the user interface of a distributed visualisation system is specified at runtime and can change automatically to reflect system or runtime modifications. With the web user interface introduced in chapter 5 (section 5.4 on page 108) such flexible interfaces are possible with our system, and this area would be worth investigating.

Another obvious area where more research would be interesting is in pushing further the use of agents in a reflective system, combining autonomic features with agents characteristics. We also did not develop much how we could use our system to render very large datasets (as other members of the e-Viz project worked on that particular aspect); several approaches described in the literature would be good candidates for implementation, notably approaches based on reflectivity would be able to leverage the existing system mechanisms.

The distributed graphic pipeline as implemented use typed data flows, but we mostly worked with image-space rendering. It would be interesting to add more dataflows such as high-level graphic commands, similar to the Chromium pipeline with OpenGL [130], but taking advantage of the introspection features and reflective capabilities of our system. Similarly, hybrid architectures such as Parallel-SG [196] could be implemented on our system in addition to our sort-first distribution mechanism.

Finally, the work presented in Chapter 8, creating and integrating performance models into the system, only dealt with simple models. While in our use-cases those simple models were useful enough, it would be interesting to add more detailed and more complex models which could compete against each other within the system. One potential approach would be to leverage the meta-information already present in the system (*i.e.* the pipeline description) to generate more detailed models automatically.

Appendices

(APPENDIX ... A)

Delphe: a Simulation Tool

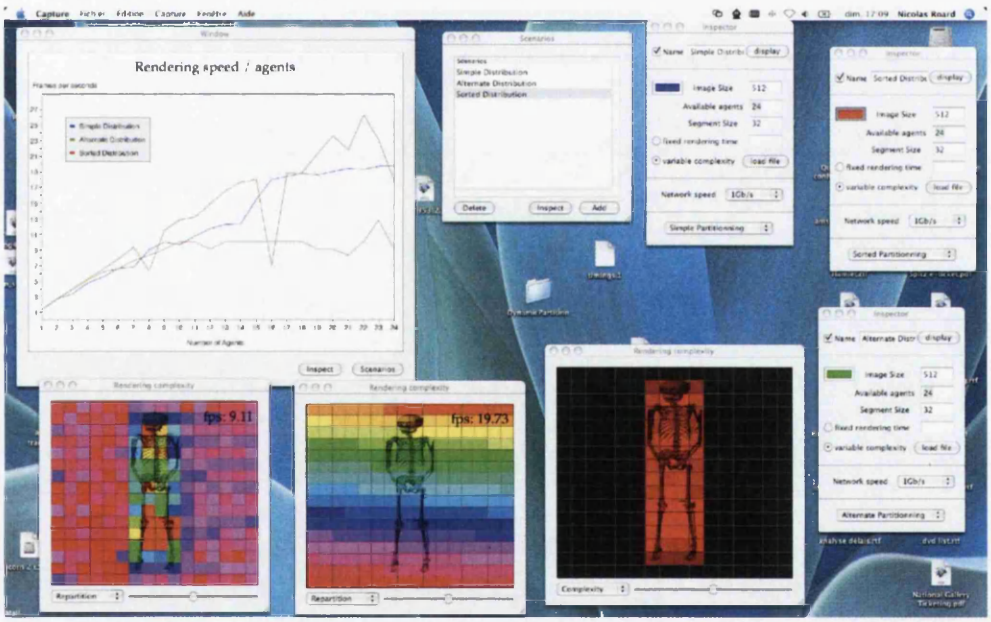


Figure A.1: *Delphe screenshot, showing a graphical representation of tiles distribution, image complexity and performance simulation using custom scenarios.*

Contents

A.4 Introduction

Delphe is an analysis and simulation tool for parallel systems. Run-time performance data is analysed and simulation algorithms can be used with the data to predict future performances. Comparing the actual performance data with its predicted best case scaling scenarios also allows users to identify bottlenecks.

Delphe can analyse either a real-time stream of data, connecting to a running system, or post-mortem data generated by a pipeline. Delphe also allows exporting the data (so far generating a set of files for gnuplot automatically, but this could be easily adapted to other formats).

A.5 Scenarios

Delphe revolves around the notion of *scenarios*, representing a performance run. Those scenarios can be a completely hypothetical run, using a simulation model, or based on real data.

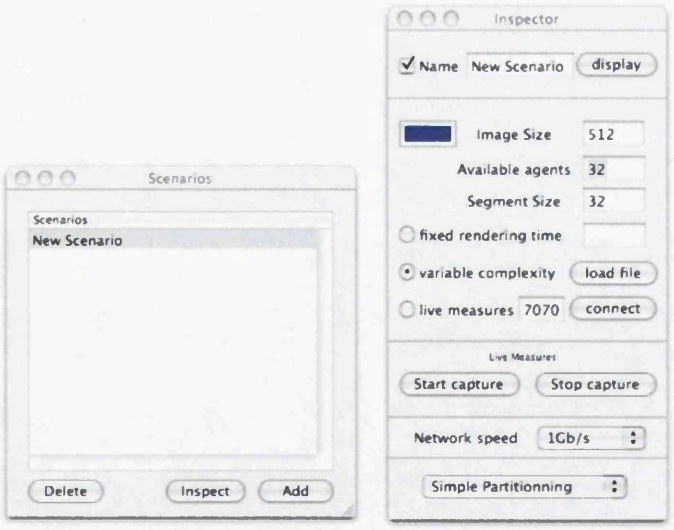


Figure A.2: *Scenarios panel and Scenario inspector*

When starting Delphe, we have the possibility of adding a new scenario. Multiple scenarios can be used together (figure A.1 on the previous page) to be compared.

A.6 Parameters

Once a scenario is created, we can open its inspector to change its parameters; most importantly we can choose the origin of the data possibly used by the scenario, as

well as more specific parameters depending on the type of scenario.

Figure A.2 on the preceding page shows an inspector for a scenario implementing a prediction model, and allowing the user to choose the type of tiles distribution.

A.7 Simulation

Using the data, a simulation scenario try to predict the future performance (figures A.3). Simulation scenarios tries to predict future performances of a system, using different approaches:

- based on historical data
- based on a simulation model of the system
- based on parametric models

Delphe simplifies trying different models against each other — in fact, one model can use other models to compute its results.

A.7.1 Performance Predictions

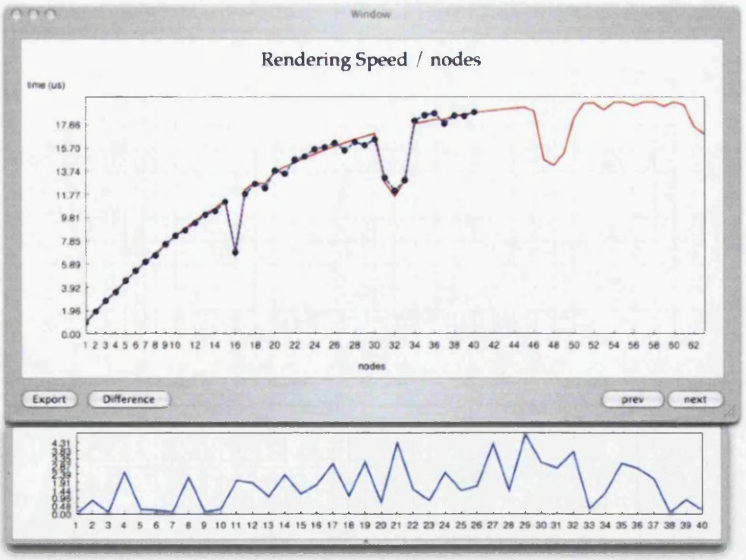


Figure A.3: *Simulation of performances and error difference between the measured data and the model*

Figure A.3 shows an example of a complex simulation model, using both a parametric and historical data model. Delphe can display a graph of the prediction, with the second graph at the bottom of the window showing the error difference between the measured data and the predicted values.

The graph generation is done through *Pointillist*, a 2D graph library we developed during the course of this project. Additional studies (such as exponential moving average) can be added to a graph.

A.7.2 Tiles Distribution and Image Complexity

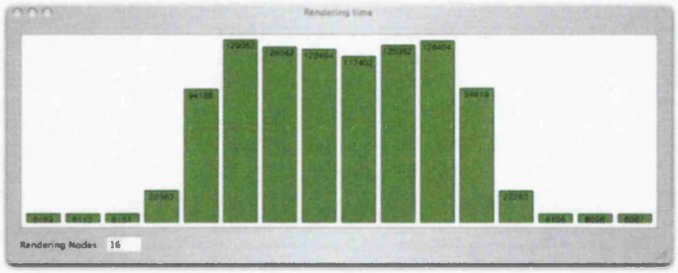


Figure A.4: *Buckets load for 16 nodes with an alternate distribution, clearly showing the load imbalance*

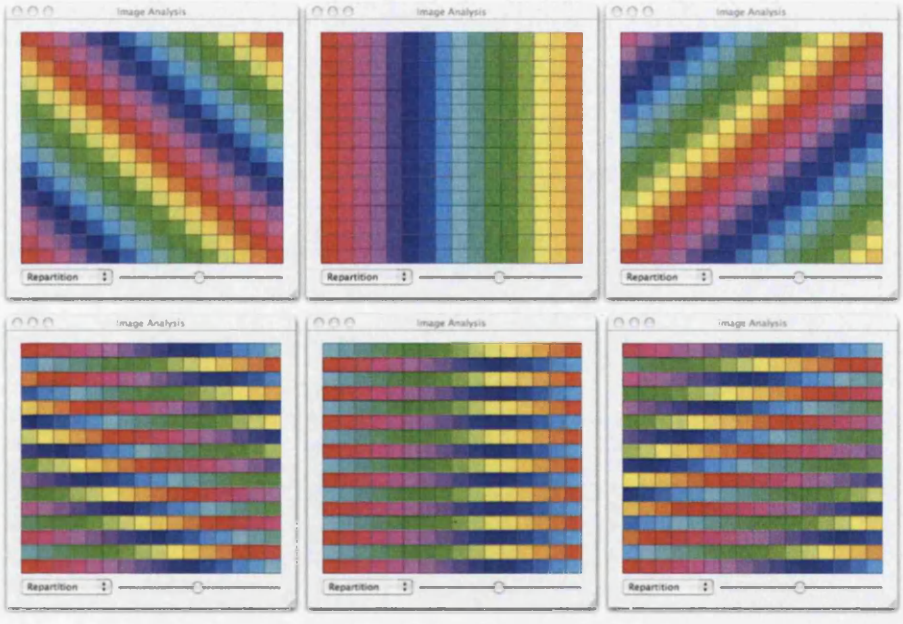


Figure A.5: *Imbalance caused by the tiles splitting mechanism. Top row respectively shows 15, 16 and 17 nodes used. Bottom row shows the tiles distribution with 30, 32 and 34 nodes. Each color corresponds to the same agent. The central column shows an alignment of agents, producing the buckets imbalance.*

Delphe can also be used to show specific information in a graphic pipeline; most notably, it is possible to show the different tiles distribution strategies used, and graphically show phenomens such as the load imbalance caused by it (figure A.5 on

the previous page). The resulting image complexity per tile can also be displayed (figure A.6).

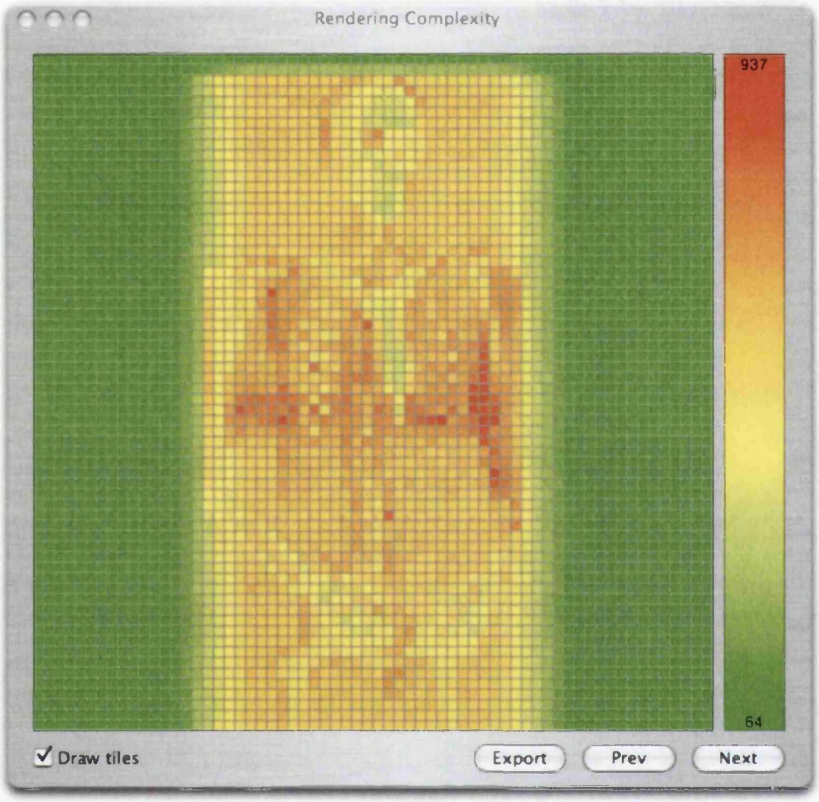


Figure A.6: *Image complexity in Delphe*

A.7.3 Load Imbalance

A final possibility while analysing a particular distribution mechanism is to display the predicted (or measured if we use real data) load imbalance. Figure A.4 on the previous page shows an example of such imbalance.

A.8 Conclusion

Delphe is a useful companion application to our visualisation system, as it provides us with additional insight on the behaviour and mechanisms of a particular pipeline. Having immediate visual feedback on performance is of particular interest. While it is obviously highly dependent on our system, as it uses our own data serialisation mechanisms to extract the data, the software has an open architecture that could be extended to accept other data formats in input or output.

(APPENDIX ...B)

Adding Smalltalk Support

"I invented the term Object-Oriented, and I can tell you I did not have C++ in mind"

— Alan Kay

Contents

B.1 Introduction

Smalltalk is an object-oriented dynamic programming language. It was created in the 1970s by a Xerox PARC team (notably composed of Alan Kay, Dan Ingalls, Adele Goldberg, Ted Kaehler and Scott Wallace), that was researching educational use of computers. Smalltalk was influenced by Lisp, Logo, Sketchpad and Simula. The original public release was called Smalltalk-80, later described in the book written by Goldberg and Robson [101].

Smalltalk is notable as a language for its message-oriented approach, its reflective features and focus on domain specific languages, and for its minimalistic syntax (famously contained fully on a postcard, figure B.1 on the following page).

Historically, Smalltalk is important for its influence; it was one of the first object-oriented languages, the Smalltalk graphical environment famously inspired the Macintosh team visiting the Xerox Parc. Due to its highly dynamic object-oriented approach and its fully reflective nature, Smalltalk was also one of the foundations for the design pattern movement.

Conceptually, one of the most important concepts of Smalltalk is that everything is an object. Objects are described via Classes (...themselves objects; objects are

instances of a class, classes being instances of metaclasses), and in terms of messages they can receive (messages that also are objects!).

The second important concept is the message-oriented nature of Smalltalk; objects send messages to other objects to do an action, rather than doing a simple static function call. This in turns allow an extremely rich flexibility, as objects can receive any messages (possibly undefined), allowing for very dynamic architectures (it is trivial to have an object acting as a bridge, receiving messages it does not understand, and forwarding them to other objects). Smalltalk is also interesting as it is close to the Actor's model [120]: extending Smalltalk to have remote messaging (*e.g.*, [34]) smoothly integrate with the language.

```
0  exampleWithNumber: x
   "A method that illustrates every part of Smalltalk
   method syntax except primitives. It has unary,
   binary, and keyword messages, declares arguments
   and temporaries, accesses a global variable (but
5  not and instance variable), uses literals (array,
   character, symbol, string, integer, float), uses
   the pseudo variables true false, nil, self, and
   super, and has sequence, assignment, return and
   cascade. It has both zero argument and one
10 argument blocks."

   |y|

   true & false not & (nil isNil) ifFalse: [self halt].
15 y ← self size + super size.
   #($a #a "a" 1 1.0)
   do: [:each | Transcript show: (each class name);
   show: ' '].
   ↑ x < y
```

Listing B.1: Smalltalk minimalistic syntax

B.2 Adding Smalltalk Support

The agent system described in this thesis is composed of many processes communicating together, most of them written in Objective-C. While reasonably easy to write new agents, adding scripting language support allowed faster iterations. Smalltalk was an obvious candidate to implement, as its language semantic is so close to Objective-C (to be more exact, it is the reverse as Objective-C was explicitly modelled after Smalltalk). Furthermore, a Smalltalk interpreter was available,

packaged with *StepTalk*, a framework for adding scripting capabilities to GNUstep and Cocoa applications, permitting the execution of simple code using provided objects. StepTalk is in fact a rather generic scripting engine: you can provide to the framework a dictionary of objects with their names, and access those objects in the script. StepTalk is not necessarily tied to Smalltalk either – other languages are supported such as Scheme (a Lisp dialect), and adding support for new languages is reasonably easy.

B.2.1 Problems

A problem with StepTalk was its focus on executing only a code snippet, *i.e.* like a function would do. The first thing was thus to create a *fake* object that would execute script snippets as methods; the object would be a normal Objective-C object, using the reflective features of the Objective-C runtime to execute a specific script when receiving a message – so that for example we could define a script method, add it to the fake Objective-C object, and when calling this method normally in Objective-C, the object would intercept the call and execute the script on the fly through StepTalk, passing itself as a "self" variable. We did implement such a system successfully, and after discussion with the StepTalk maintainer this exact mechanism was later implemented by him through a *STActor* class.

B.2.2 Loading Smalltalk Code

Now that StepTalk supported the creation of proxy objects behaving like normal objects from the other Objective-C objects point of view, it was easy to modify the Objective-C frameworks we wrote (implementing the mechanisms to create an agent) to take advantage of using the STActor class. In that way we could create a normal EAgent (our main Objective-C class implementing agent behaviour) subclass using STActor, and we wrote a simple application (*AgentInterpreter*) to do just that; this application would then load a file and parse it to create the STActor instance it needed.

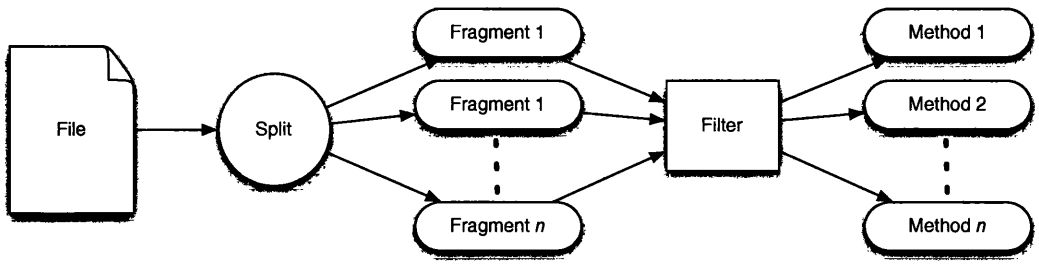


Figure B.7: *Smalltalk code loading*

Figure B.7 on the previous page shows the general loading mechanism of a script by *AgentInterpreter*; the file is loaded, then splitted in different fragments, using the file-out Smalltalk convention of separating the methods by a "!!" string. The fragments would then be considered as Smalltalk methods for the object corresponding to the loaded file.

```
0  #!/usr/local/bin/AgentInterpreter

   !!

   initialize
5      agentName ← 'WordCount␣pipeline␣example'.
       agentType ← 'Pipeline'.

   !!

10 domain
       read ← Agent ofType: 'FileReader' named: 'reader'.
       split ← Agent ofType: 'StringSplitter' named: 'split'.
       count ← Agent ofType: 'WordCounter' named: 'count'.
       result ← Agent ofType: 'Reduce' named: 'total'.
15
       read -: 'string' :↦ split.
       split -: 'array' :↦ count.
       count -: 'number' :↦ result.

20     self add: read.
       self add: split.
       self add: count.
       self add: result.

25 !!

   start
       (self object: 'reader') start.
```

Listing B.2: Pipeline agent written using a Smalltalk script

Finally, we added a simple filtering step, to simplify writing pipeline agents, replacing some specific strings by others:

- "-:" is transformed into "output:"
- ":->" is transformed into "to:"
- "->" is transformed into "outputTo:"

Figure B.2 on the preceding page shows an example of a simple pipeline created through the scripting capabilities. The *domain* method is automatically called by the framework and allows to specify an execution context, here defining the relationships between different agents. Corresponding agents to this model are created on demand.

(APPENDIX ...C)

User Interface: Steering widget

Contents

Introduction

While this thesis does not concern user interfaces, we developed a novel steering widget that we used in our system in a variety of incarnations, notably for volume dataset visualisations. We present in this annex the widget and some use cases and possible future developments.

C.3 Interaction Principle

When visualising a dataset, the user has to set the camera viewpoint, as well as the look-at point. Many different approaches exist – as crude as using sliders representing the camera 3 dimensional position, or as advanced as first-person view. As the parameter space to set a viewport is large, a usual solution is to constrain the possible positions; *e.g.* a first-person view would set the look-at point by default (as looking forward in the same direction as the movement), possibly constraining the movement of the character to a 2D plane, etc.

In our case, we focused on visualising volume datasets; the type of interactions we ended up doing mostly consisted of turning around the volume, zooming in and out, and up and down.

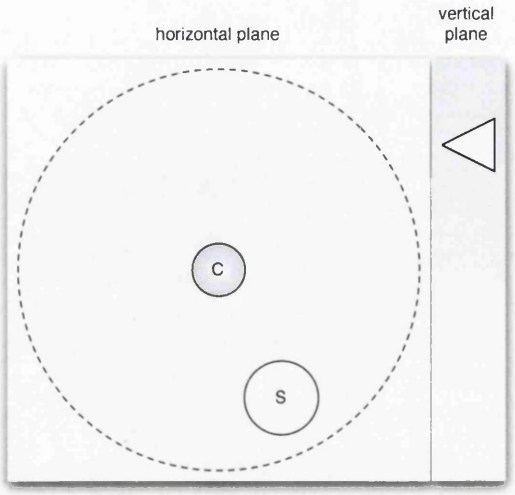


Figure C.8: *Steering widget*

We then created a widget simplifying those interactions while working on a mobile device with touchscreen (figure C.10 on the next page). Figure C.8 shows the general design of the widget:

- the main part (containing the circle area) represents a slice of a volume, seen from above; the right part is a slider controlling our actual "height" or slice position.
- a central point *C* represents the look at point; the user can click on it and move it around (although the default central position is usually the correct one)
- a circle *S* represents the position of the user; clicking and dragging it will modify the user's camera position
- the large dotted grey circle represents a movement constraint: the user cannot move *C* or *S* outside this area

The result of this interaction model is to constrain the movements in a 2D plane, allowing to turn around a volume dataset and to easily explore it by zooming. It must be noted that more complex navigation is probably not a good fit for this particular steering widget (*e.g.* the look-at point is fixed and requires two steps to be moved; this is rather inefficient).

C.4 Implementations

We principally used this widget in two incarnations: first in the Squeak Smalltalk environment, when we started prototyping our visualisation environment (figure C.9 on the next page), then as a Java applet for our web visualisation client (figure C.4.2

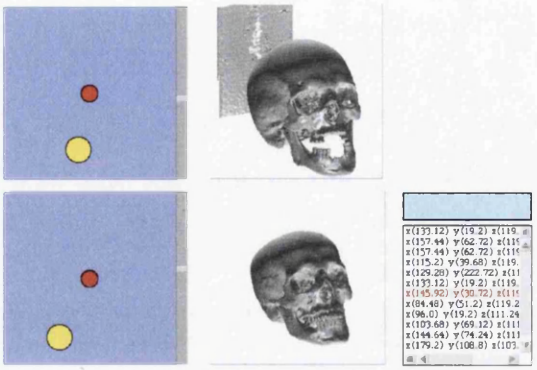


Figure C.9: *Example of the steering widget running in Squeak*

on the following page).

C.4.1 Squeak

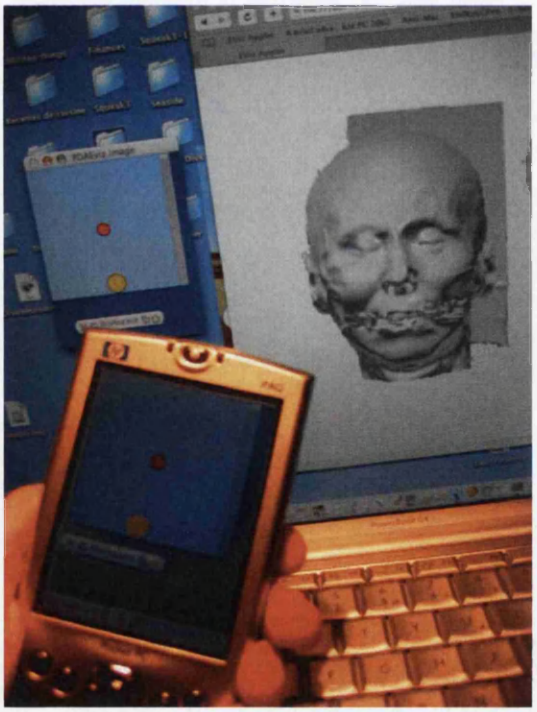


Figure C.10: *Using the steering widget on a mobile device as a remote control*

The Squeak implementation was the first one, and was done using the *morphix* tookit, allowing us to create ad hoc UI (a visualisation surface was also available) as shown on figure C.9, where two linked visualisation pipelines are displayed.

A side effect of having this implementation is that it was easy to demonstrate

a working prototype on a mobile device, as a Squeak image (*i.e.* our code) is platform-independent and a Virtual Machine is available on Windows Mobile devices. Figure C.10 on the preceding page shows an example of using a mobile device connected to a remote visualisation pipeline, with a visualisation client running on another computer. One use case of mobile devices we found interesting was to use them as remote controls rather than full visualisation clients; as a visualisation client, a mobile device is rather poor due to its small screen estate and typically high latency network connection, but as a remote control it can be particularly powerful.

C.4.2 Java

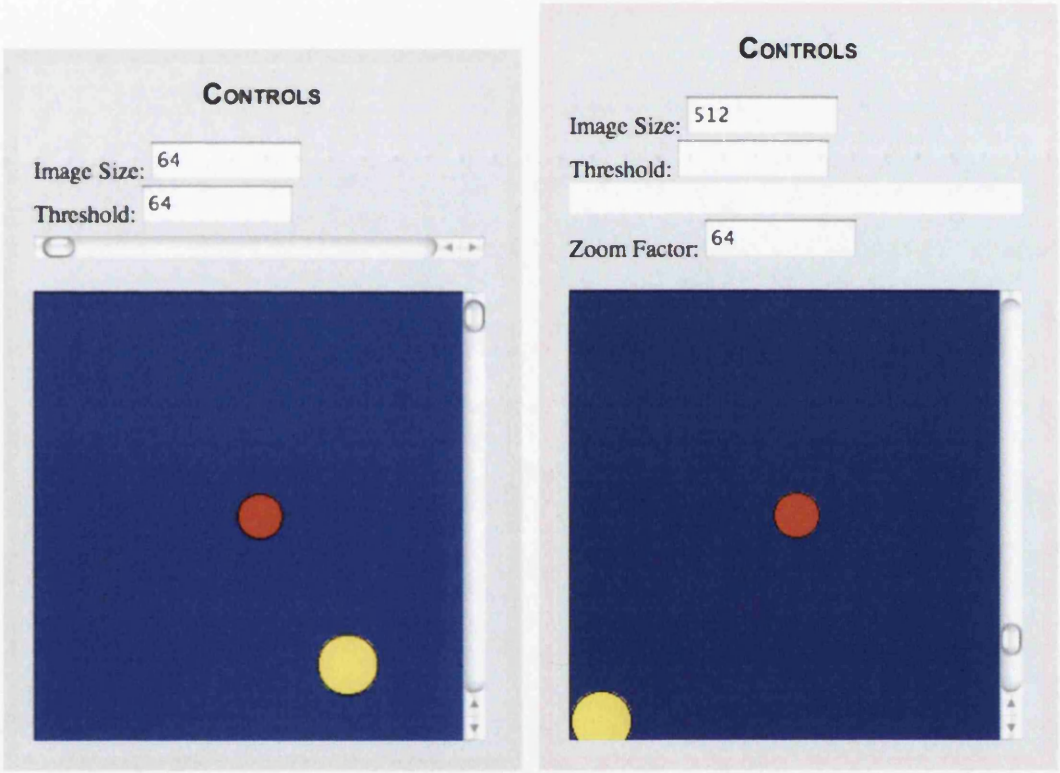


Figure C.11: *The steering widget used in the web client*

The second implementation we wrote was a Java applet; this let us create complex user interfaces leveraging web technologies. Most notably, such a user interface is much more flexible than typical clients. Figure C.4.2 shows the steering widget part of a web user interface, and we can see that additional pipeline parameters can easily be added (Zoom factor textfield).

C.4.3 Cocoa/OpenStep and iPhone

We also ported the widget to OpenStep, and were thus able to use it on MacOS X. With some additional efforts, we were able to use the widget on the iPhone platform, along with a visualisation client. Notably, we had to use OpenGL ES as the iPhone UI framework presented a bottleneck when displaying both the steering widget and a visualisation surface.

C.5 Future Development

During the development of the system, we used the Squeak implementation to run a visualisation client on a Windows CE device. A notable use case was to use the PDA as a remote control (see figure C.10 on page 201) of a visualisation session running on a different computer.

Visualising directly on the mobile device is another compelling use case, but the physically small screen and lack of good controls make it a poor user experience. We prototyped a custom version of the client that displayed both a visualisation surface and the control steering widget (figure C.12 on the left).

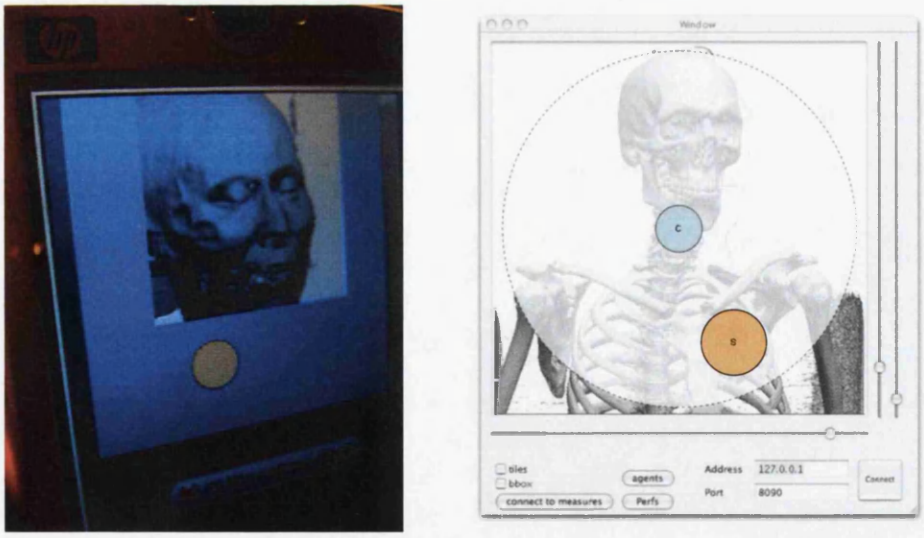


Figure C.12: *In-image steering interaction*

While working reasonably well, and an improvement over a lack of displayed controls, the interaction was not really polished. Hiding completely the controls (but still responding to pen strokes) was better as the visualisation surface could be full screen, but lacked the ease of use of the steering widget. A possible solution that could be implemented in the future would be to have a fullscreen display surface, but if the user touches the screen with the stylus, the steering control widget would

appear semi-transparently. Figure C.12 on the preceding page on the right shows a mockup of what the UI would look like.

C.5.1 Additional Platforms

While the widget is reasonably easy to create (and thus to port to a new environment), we mostly used it within Squeak (for prototyping) and as a Java applet (used with the web client). While the Java applet works perfectly well, it suffers the usual problems of an applet; mainly, security issues and slow startup.

The web, as an application platform, is improving constantly; new developments such as the HTML 5 specification greatly enhance the potential of a javascript+html (+svg ?) platform. As such, porting the widget to javascript, in a web perspective, would be compelling, as it would render the applet expandable, and the user interface would be accessible on a larger number of platforms (any modern browser, *i.e.* platforms such as the Apple iPhone, Google's Android, or Nokia mobile devices would work transparently). In the same perspective of the web as a platform, other potentially interesting targets would be the Microsoft Silverlight platform, the Adobe AIR platform, and for Windows Mobile devices, an ActiveX control.

In addition to a simple HTML-based image client (*i.e.* with a pipeline agent acting as an HTTP server delivering an image), this would allow the creation of a cross-platform visualisation client that would work on nearly any modern platform without modifications.

References

[1] Advanced Visual Systems (AVS). URL: http://www.avs.com, Last checked: *December 2007.*

[2] Covise. URL: http://www.visenso.de/index.php?id=148&L=1, Last checked: *December 2007.*

[3] CUMULVS. URL: http://acts.nersc.gov/cumulvs/, Last checked: *December 2007.*

[4] IRIS Explorer.

[5] OpenDX.

[6] OpenMP. , URL: http://www.openmp.org/.

[7] pV3 (parallel Visual3). URL: http://raphael.mit.edu/pv3/pv3.html, Last checked: *December 2007.*

[8] VNC. URL: http://www.realvnc.com, Last checked: *December 2007.*

[9] URL: http://www.distributed.net, (1997), Last checked: *December 2007.*

[10] OpenRM Scene Graph. URL: http://openrm.sourceforge.net, (2000).

[11] M. ABD-EL-MALEK, G. R. GANGER, G. R. GOODSON, M. K. REITER & J. J. WYLIE. Fault-scalable Byzantine fault-tolerant services. *SIGOPS Oper. Syst. Rev.* 39(5), 59–74 (2005), DOI: http://doi.acm.org/10.1145/1095809.1095817.

[12] N. ADIGA *et al.* An overview of the BlueGene/L Supercomputer. in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, 1–22 (IEEE Computer Society Press, Los Alamitos, CA, USA, 2002).

[13] M. AGARWAL *et al.* Automate: Enabling Autonomic Grid Applications. *Cluster Computing: The Journal of Networks, Software Tools, and Applications – Special Issue on Autonomic Computing, Kluwer Academic Publishers* 9(1) (2006).

[14] A. A. AHMED, M. S. A. LATIFF, K. A. BAKAR & Z. A. RAJION. Visualization Pipeline for Medical Datasets on Grid Computing Environment. in *ICCSA '07: Proceedings of the The 2007 International Conference Computational Science and its Applications*, 567–576 (IEEE Computer Society, Washington, DC, USA, 2007).

[15] J. AHRENS *et al.* Large-scale data visualization using parallel data streaming. *IEEE CG&A* 21(4), 34–41 (2001).

[16] G. AMDAHL. Validity of the single-processor approach to achieving large scale computing capabilities. in *AFPIS Conference Proceedings*, (V. AFIPS Press, Reson, ed), volume 30, 483–485, (1967).

[17] D. P. ANDERSON. BOINC: A System for Public-Resource Computing and Storage. in *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, 4–10 (IEEE Computer Society, Washington, DC, USA, 2004).

[18] D. P. ANDERSON, J. COBB, E. KORPELA, M. LEBOFSKY & D. WERTHIMER. SETI@home: an experiment in public-resource computing. *Commun. ACM* 45(11), 56–61 (2002), DOI: `http://doi.acm.org/10.1145/581571.581573`.

[19] G. R. ANDREWS *et al.* An overview of the SR language and implementation. *ACM Trans. Program. Lang. Syst.* 10(1), 51–86 (1988), DOI: `http://doi.acm.org/10.1145/42192.42324`.

[20] APPLE. Cocoa Framework. URL: `http://developer.apple.com`, (1999).

[21] J. ARMSTRONG. The development of Erlang. in *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, 196–203 (ACM, New York, NY, USA, 1997).

[22] J. ARMSTRONG. *Making reliable distributed systems in the presence of software errors.* PhD thesis, The Royal Institute of Technology, Department of Microelectronics and Information Technology, (2003).

[23] J. ARMSTRONG. A history of Erlang. in *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 6–1–6–26 (ACM, New York, NY, USA, 2007).

[24] W. C. ARNOLD, D. W. LEVINE & E. C. SNIBLE. Autonomic Manager Toolkit. (2003).

[25] M. ASTLEY, D. C. STURMAN & G. A. AGHA. Customizable middleware for modular distributed software. *Communications of the ACM* 44(5), 99–107 (2001).

[26] J. W. BACKUS *et al. Fortran: Automatic Coding System for the IBM 704 EDPM.* (1956).

[27] S. BAKER. CORBA Distributed Objects. *Addison-Wesley* (1997).

[28] G. BELL, A. PARISI & M. PESCE. VRML 1.0. , URL: http://www.web3d.org/x3d/specifications/vrml/VRML1.0/index.html.

[29] E. W. BETHEL, G. HUMPHREYS, B. PAUL & J. D. BREDERSON. Sort-first, distributed memory parallel visualization and rendering. *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics* , 41–50 (2003).

[30] E. W. BETHEL & J. SHALF. Grid-Distributed Visualizations Using Connectionless Protocols. *IEEE Comput. Graph. Appl.* 23(2), 51–59 (2003), DOI: http://dx.doi.org/10.1109/MCG.2003.1185580.

[31] A. D. BIRRELL & B. J. NELSON. Implementing remote procedure calls. *ACM Trans. Computer Systems* , 39–59 (1984).

[32] R. BJORNSON, N. CARRIERO, D. GELERNTER & J. LEICHTER. Linda in adolescence. in *EW 2: Proceedings of the 2nd workshop on Making distributed systems work*, 1–4 (ACM, New York, NY, USA, 1986).

[33] G. BOOCH. Object-Oriented Analysis and Design with Applications. *Addison-Wesley, Reading, MA* (1994).

[34] J.-P. BRIOT. From objects to actors: study of a limited symbiosis in smalltalk-80. in *Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*, 69–72 (ACM, New York, NY, USA, 1988).

[35] K. BRODLIE *et al.* Visual supercomputing: Technologies, application and challenges. *Computer Graphics Forum* 24(2), 217–245 (2005).

[36] K. BRODLIE *et al.* e-Viz. URL: http://www.eviz.org/, Last checked: *December 2007*.

[37] K. BRODLIE *et al.* GRASPARC: a problem solving environment integrating computation and visualization. in *VIS '93: Proceedings of the 4th conference on Visualization '93*, 102–109, (1993).

[38] K. BRODLIE *et al.* Adaptive Infrastructure for Visual Computing. in *Theory and Practice of Computer Graphics*, 147–156, (2007).

[39] J. M. BROOKE *et al.* Computational steering in RealityGrid. *Proc. UK e-Science All Hands Meeting* (2003).

[40] D. BUAKLEE, G. F. TRACY, M. K. VERNON & S. J. WRIGHT. Near-optimal adaptive control of a large grid application. in *ICS '02: Proceedings of the 16th international conference on Supercomputing*, 315–326 (ACM, New York, NY, USA, 2002).

[41] R. BUYYA. High Performance Cluster Computing, Vol. 2: Programming and Applications. *Prentice Hall* (1999).

[42] B. CABRAL, N. CAM & J. FORAN. Accelerated volume rendering and tomo-

graphic reconstruction using texture mapping hardware. *Proc. ACM/IEEE Symp. Volume Visualization* , 91–98 (1995).

[43] G. CAMERON. Modular Visualization Environments: Past, Present and Future. *Computer Graphics* 29(2), 3–4 (1995).

[44] N. CARRIERO & D. GELERNTER. The S/Net's Linda kernel. *ACM Trans. Comput. Syst.* 4(2), 110–129 (1986), DOI: `http://doi.acm.org/10.1145/214419.214420`.

[45] N. CARRIERO, D. GELERNTER & J. LEICHTER. Distributed data structures in Linda. in *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 236–242 (ACM, New York, NY, USA, 1986).

[46] N. CARVER & V. LESSER. The Evolution of Blackboard Control Architectures. Technical Report UM-CS-1992-071, University of Massachusetts, , (1992).

[47] M. CASTRO & B. LISKOV. Proactive recovery in a Byzantine-fault-tolerant system. in *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, 19–19 (USENIX Association, Berkeley, CA, USA, 2000).

[48] M. CASTRO & B. LISKOV. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.* 20(4), 398–461 (2002), DOI: `http://doi.acm.org/10.1145/571637.571640`.

[49] V. CERF, Y. DALAL & C. SUNSHINE. SPECIFICATION OF INTERNET TRANSMISSION CONTROL PROGRAM. URL: `http://tools.ietf.org/html/rfc675`, December (1974), Last checked: *December 2007.*

[50] T. D. CHANDRA & S. TOUEG. Unreliable failure detectors for reliable distributed systems. *J. ACM* 43(2), 225–267 (1996), DOI: `http://doi.acm.org/10.1145/226643.226647`.

[51] K. M. CHANDY & C. KESSELMAN. Compositional C++: Compositional Parallel Programming. in *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, 124–144 (Springer-Verlag, London, UK, 1993).

[52] C.-C. CHANG, G. CZAJKOWSKI, T. VON EICKEN & C. KESSELMAN. Evaluating the performance limitations of MPMD communication. in *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, 1–10 (ACM, New York, NY, USA, 1997).

[53] S.-W. CHENG, A.-C. HUANG, D. GARLAN, B. SCHMERL & P. STEENKISTE. An Architecture for Coordinating Multiple Self-Management Systems. in *Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, (2004).

[54] C. H. CHIEN & J. K. AGGARWAL. Volume/Surface octrees for the representation of three-dimensional objects. *Computer Vision, Graphics and Image Processing* 36(1), 100–113 (1986).

[55] D. CHISNALL & M. CHEN. The Making of SimEAC. in *International Conference on Autonomic Computing*, 301–302, (2006).

[56] D. CHISNALL, M. CHEN & C. HANSEN. Knowledge-based out-of-core algorithms for data management in visualisation. *Proc. EuroVis 2006* , 107–114 May (2006).

[57] E. CHRISTENSEN, F. CURBERA, G. MEREDITH & S. WEERAWARANA. Web Services Description Language (WSDL) 1.1. URL: `http://www.w3.org/TR/wsdl`, Last checked: *December 2007.*

[58] I.-H. CHUNG & J. K. HOLLINGSWORTH. Using Information from Prior Runs to Improve Automated Tuning Systems. in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 30 (IEEE Computer Society, Washington, DC, USA, 2004).

[59] B. J. COX. Object Oriented Programming: an Evolutionary Approach. *Addison-Wesley* ISBN 0-201-54834-8 (1991).

[60] C. H. CRAWFORD & A. DAN. eModel: Addressing the Need for a Flexible Modeling Framework in Autonomic Computing. in *MASCOTS '02: Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, 203 (IEEE Computer Society, Washington, DC, USA, 2002).

[61] D. CROCKFORD. JSON. URL: `http://www.json.org`, (1999).

[62] D. CULLER *et al.* LogP: towards a realistic model of parallel computation. *SIGPLAN Not.* 28(7), 1–12 (1993), DOI: `http://doi.acm.org/10.1145/173284.155333`.

[63] D. CUTTING. Hadoop. URL: `http://wiki.apache.org/lucene-hadoop/`, (2005).

[64] K. CZAJKOWSKI *et al.* The WS-Resource Framework. *Globus whitepaper* (2004).

[65] L. DAGUM & R. MENON. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering* 05(1), 46–55 (1998), DOI: `http://doi.ieeecomputersociety.org/10.1109/99.660313`.

[66] W. J. DALLY & A. A. CHIEN. Object-oriented concurrent programming in CST. *SIGPLAN Not.* 24(4), 28–31 (1989), DOI: `http://doi.acm.org/10.1145/67387.67392`.

[67] B. D'AMORA & F. BERNARDINI. Pervasive 3D viewing for product data management. *IEEE CG&A* 23(2), 14–19 (2003).

[68] F. DAREMA. The SPMD Model: Past, Present and Future. in *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 1 (Springer-Verlag, London, UK, 2001).

[69] K. DAVIS *et al.* A Performance and Scalability analysis of the BlueGene/L architecture. *Proc. ACM/IEEE Conf. on Supercomputing* , 41–ff (2004).

[70] J. DEAN & S. GHEMAWAT. MapReduce: Simplified Data Processing on Large Clusters. *in OSDI'04, 6th Symposium on Operating Systems Design and Implementation, Sponsored by USENIX, in cooperation with ACM SIGOPS* , 137–150 (2004).

[71] M. DEERING & D. NAEGLE. The SAGE graphics architecture. in *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, 683–692 (ACM, New York, NY, USA, 2002).

[72] L. P. DEUTSCH. The Eight Fallacies of Distributed Computing. URL: `http://blogs.sun.com/jag/resource/Fallacies.html`, (1991), Last checked: *December 2007.*

[73] L. J. DOCTOR & J. G. TORBORG. Display techniques for octree-encoded objects. *IEEE CG&A* 3(1), 29–38 (1981).

[74] D. DUKE, M. WALLACE, R. BORGO & C. RUNCIMAN. Fine-grained Visualization Pipelines and Lazy Functional Languages. *IEEE Transactions on Visualization and Computer Graphics* 12(5), 973–980 (2006), DOI: `http://dx.doi.org/10.1109/TVCG.2006.145`.

[75] S. D. DYER. Visualization: a dataflow toolkit for visualization. *IEEE Computer Graphics and Applications* 10(4), 60–64 (1990).

[76] E. N. M. ELNOZAHY, L. ALVISI, Y.-M. WANG & D. B. JOHNSON. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.* 34(3), 375–408 (2002), DOI: `http://doi.acm.org/10.1145/568522.568525`.

[77] R. ELWALD & L. MASS. A high performance graphics system for the Cray-1. *ACM SIGGRAPH Computer Graphics* , 82–86 (1978).

[78] K. ENGEL, R. WESTERMANN & T. ERTL. Isosurface extraction techniques for web-based volume visualization. *Proc. IEEE Visualization* , 139–146 (1999).

[79] D. C. ENGELBART. A conceptual framework for the augmentation of man's intellect (Reprint). , 35–65 (1988).

[80] P. FELBER, X. DÉFAGO, R. GUERRAOUI & P. OSER. Failure Detectors as First Class Objects. in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99)*, 132–141 (, Edinburgh, Scotland, 1999).

[81] P. FELDMAN & S. MICALI. Optimal algorithms for Byzantine agreement. in *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, 148–161 (ACM, New York, NY, USA, 1988).

[82] T. Y. FENG. A survey of interconnection networks. *IEEE Computers* , 12–27 (1981).

[83] R. T. FIELDING. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, (2000).

[84] T. FININ, R. FRITZSON, D. MCKAY & R. MCENTIRE. KQML as an agent communication language. in *CIKM '94: Proceedings of the third international conference on Information and knowledge management*, 456–463 (ACM, New York, NY, USA, 1994).

[85] M. J. FISCHER, N. A. LYNCH & M. S. PATERSON. Impossibility of distributed consensus with one faulty process. *J. ACM* 32(2), 374–382 (1985), DOI: `http://doi.acm.org/10.1145/3149.214121`.

[86] M. FLYNN. Very High-Speed Computing Systems. *Proceedings of the IEEE* 54(12), pp. 1901–1909 December (1966).

[87] M. FLYNN. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers* C-21(9), pp. 948–960 september (1972).

[88] S. FORTUNE & J. WYLLIE. Parallelism in random access machines. *Proc. ACM Symp. Theory of Computing* , 114–118 (1978).

[89] I. FOSTER & C. KESSELMAN. Globus: A metacomputing infrastructure toolkit. *Int. J. Supercomputer Applications* 11(2), 115–128 (1997).

[90] I. FOSTER & C. KESSELMAN. The Globus project: a status report. *Proc. Heterogeneous Computing Workshop* , 4–18 (1998).

[91] I. FOSTER & C. KESSELMAN. The Grid: Blueprint for a New Computing Infrastructure. *Morgan Kaufmann* (1999).

[92] I. FOSTER, C. J. N. KESSELMAN & S. TUECKE. The physiology of the Grid: an open Grid services architecture for distributed system integration. *Open Grid Services Infrastructure WG* (2002).

[93] I. FOSTER. Globus toolkit version 4: Software for service-oriented systems. *Proc. of the IFIP International Conference on Network and Parallel Computing* , 2–13 (2005).

[94] FREE SOFTWARE FOUNDATION. GNUstep, a GNU implementation of the OpenStep specifications. URL: `http://www.gnustep.org`, (1994).

[95] W. FU & Q. HUANG. GridEye: A Service-oriented Grid Monitoring System with Improved Forecasting Algorithm. in *GCCW '06: Proceedings of the Fifth*

International Conference on Grid and Cooperative Computing Workshops, 5–12 (IEEE Computer Society, Washington, DC, USA, 2006).

[96] H. FUCHS, Z. M. KEDEM & B. F. NAYLOR. On visible surface generation by a priori tree structures. in *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, 124–133 (ACM, New York, NY, USA, 1980).

[97] G. A. GEIST, A. BEGUELIN, J. DONGARRA, W. JIANG & R. M. AN V. SUNDERAM. PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing. *MIT Press* (1994).

[98] G. A. GEIST, J. A. KOHLA & P. M. PAPADOPOULOUS. PVM and MPI: A comparison of features. *Calculateurs Paralleles* 8(2), 137–150 (1996).

[99] D. GELERNTER. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.* 7(1), 80–112 (1985), DOI: `http://doi.acm.org/10.1145/2363.2433`.

[100] A. GHODSI & J. ARMSTRONG. URL: `http://www.sics.se/~joe/apachevsyaws.html`, Last checked: *December 2007*.

[101] A. GOLDBERG & D. ROBSON. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, (1983).

[102] J. B. GOODENOUGH. Exception handling: issues and a proposed notation. *Commun. ACM* 18(12), 683–696 (1975), DOI: `http://doi.acm.org/10.1145/361227.361230`.

[103] J. GRAY. The transaction concept: virtues and limitations (invited paper). in *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases*, 144–154. VLDB Endowment, (1981).

[104] S. GREEN. Parallel Processing for Computer Graphics. *MIT Press* (1991).

[105] I. J. GRIMSTEAD, N. J. AVIS & D. W. WALKER. Automatic Distribution of Rendering Workloads in a Grid Enabled Collaborative Visualization Environment. in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 1 (IEEE Computer Society, Washington, DC, USA, 2004).

[106] I. J. GRIMSTEAD, N. J. AVIS & D. W. WALKER. Visualization across the pond: How a wireless PDA can collaborate with million-polygon datasets via 9,000km of cable. *Web3d '05: Proceedings of the tenth international conference on 3D Web technology* , 47–56 (2005).

[107] M. GUDGIN *et al.* Simple Object Access Protocol (SOAP) 1.2. , URL: `http://www.w3.org/TR/soap12`.

[108] Z. GUESSOUM, N. FACI & J.-P. BRIOT. Adaptive replication of large-scale multi-agent systems – towards a fault-tolerant multi-agent platform. *Proceed-*

ings of the 4th International Workshop on Software Engineering for Large-Scale multi-agent systems , 1–6 (2005).

[109] N. J. GUNTHER. A New Interpretation of Amdahl's Law and Geometric Scalability, (2002).

[110] N. J. GUNTHER. Unification of Amdahl's Law, LogP and Other Performance Models for Message-Passing Architectures. in *IASTED PDCS*, 569–576, (2005).

[111] N. J. GUNTHER. A General Theory of Computational Scalability Based on Rational Functions. *CoRR* abs/0808.1431 (2008).

[112] J. L. GUSTAFSON, G. R. MONTRY & R. E. BENNER. Development of parallel methods for a 1024 hypercubes. *SIAM J. Scientific and Statistical Computing* 9(4) (1988).

[113] J. L. GUSTAFSON. Reevaluating Amdahl's Law. in *Communications of the ACM*, volume 31, 532–533, (1988).

[114] R. HABER & D. MCNABB. Visualization idioms: A conceptual model for scientific visualization systems. *Visualization in Scientific Computing, IEEE Computer Society Press* (1990).

[115] P. HARTLING, A. BIERBAUM & C. CRUZ-NEIRA. Virtual reality interfaces using Tweek. *In SIGGRAPH 2002 Sketches* (2002).

[116] N. HAYASHIBARA, A. CHERIF & T. KATAYAMA. Failure Detectors for Large-Scale Distributed Systems. in *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, 404 (IEEE Computer Society, Washington, DC, USA, 2002).

[117] B. HAYES-ROTH. A blackboard architecture for control. *Artif. Intell.* 26(3), 251–321 (1985), DOI: `http://dx.doi.org/10.1016/0004-3702(85)90063-3`.

[118] A. HEIRICH & L. MOLL. Scalable distributed visualization using off-the-shelf components. in *PVGS '99: Proceedings of the 1999 IEEE symposium on Parallel visualization and graphics*, 55–59 (IEEE Computer Society, Washington, DC, USA, 1999).

[119] J. HENRY C. BAKER & C. HEWITT. The incremental garbage collection of processes. in *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, 55–59 (ACM, New York, NY, USA, 1977).

[120] C. HEWITT, P. BISHOP & R. STEIGER. A Universal Modular Actor Formalism for Artificial Intelligence. *IJCAI* , 235–245 (1973).

[121] C. HEWITT *et al.* Actor induction and meta-evaluation. in *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 153–168 (ACM, New York, NY, USA, 1973).

[122] W. D. HILLIS. *The connection machine.* MIT Press, Cambridge, MA, USA, (1986).

[123] W. D. HILLIS & L. W. TUCKER. The CM-5 Connection Machine: a scalable supercomputer. *Commun. ACM* 36(11), 31–40 (1993), DOI: http://doi.acm.org/10.1145/163359.163361.

[124] C. A. R. HOARE. Communicating sequential processes. *Commun. ACM* 21(8), 666–677 (1978), DOI: http://doi.acm.org/10.1145/359576.359585.

[125] I. V. D. HOOGEN. Deutsch's Fallacies, 10 Years After. URL: http://java.sys-con.com/read/38665.htm, (2004), Last checked: *December 2007.*

[126] P. HORN. Autonomic Computing: IBM Perspective on the State of Information Technology, (2001).

[127] C. HUGHES & N. JOHN. A flexible approach to high performance visualization enabled augmented reality. *Theory and Practice of Computer Graphics* , 181–186 (2007).

[128] C. HUGHES, N. JOHN & M. RIDING. A generic approach to high performance visualization enabled augmented reality. *UK e-Science All Hands Meeting 2006* , 441–444 (2006).

[129] G. HUMPHREYS *et al.* WireGL: a scalable graphics system for clusters. *Proc. ACM SIGGRAPH* (2001).

[130] G. HUMPHREYS *et al.* Chromium: a stream-processing framework for interactive rendering on clusters. *Proc. ACM SIGGRAPH* (2002).

[131] S. HWANG & C. KESSELMAN. A Flexible Framework for Fault Tolerance in the Grid. *J. GRID COMP* 1(3), 251–272 (2003).

[132] IBM. Autonomic Computing Toolkit. URL: http://www.ibm.com/developerworks/autonomic/overview.html, (2004), Last checked: *December 2007.*

[133] IBM. An architectural blueprint for autonomic computing. URL: http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf, (2004 (updated jun 2006)).

[134] J. D. ICHBIAH. Preliminary Ada reference manual. *SIGPLAN Not.* 14(6a), 1–145 (1979), DOI: http://doi.acm.org/10.1145/956650.956651.

[135] J. D. ICHBIAH *et al.* Rationale for the design of the Ada programming language. *SIGPLAN Not.* 14(6b), 1–261 (1979), DOI: http://doi.acm.org/10.1145/956653.956654.

[136] D. INGALLS, T. KAEHLER, J. MALONEY, S. WALLACE & A. KAY. Back to the future: the story of Squeak, a practical Smalltalk written in itself. in *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented*

programming, systems, languages, and applications, 318–326 (ACM, New York, NY, USA, 1997).

[137] S. IZADI, P. COUTINHO, T. RODDEN & G. SMITH. The FUSE platform: supporting ubiquitous collaboration within diverse mobile environments. *Automated Software Engineering* 9(2), 167–186 (2002).

[138] N. R. JENNINGS. On agent-based software engineering. *Artificial Intelligence* 117, 277–296 (2000).

[139] P. JOGALEKAR & M. WOODSIDE. Evaluating the scalability of distributed systems. *IEEE Trans. Parallel and Distributed Systems* 11(6), 589–603 (2000).

[140] H. JUN *et al.* A new rational model of agent for autonomic computing. *Proceedings of the 2004 IEEE International Conference on Systems, Man and Cybernetics* 6, 5531–5536 (2004).

[141] N. H. KAPADIA, J. A. B. FORTES & C. E. BRODLEY. Predictive Application-Performance Modeling in a Computational Grid Environment. in *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, 6 (IEEE Computer Society, Washington, DC, USA, 1999).

[142] A. H. KARP & H. P. FLATT. Measuring parallel processor performance. *Commun. ACM* 33(5), 539–543 (1990), DOI: `http://doi.acm.org/10.1145/78607.78614`.

[143] A. KAY *et al.* Croquet. URL: `http://www.opencroquet.org/`, (2001), Last checked: *December 2007*.

[144] J. O. KEPHART & D. M. CHESS. The Vision of Autonomic Computing. *Computer* 36(1), 41–50 January (2003), URL: `http://portal.acm.org/citation.cfm?id=642200`, DOI: `10.1109/MC.2003.1160055`.

[145] S. KIRTANE & J. MARTIN. Application performance prediction in autonomic systems. in *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, 566–572 (ACM, New York, NY, USA, 2006).

[146] F. KON, F. COSTA, G. BLAIR & R. H. CAMPBELL. The case for reflective middleware. *Communications of the ACM* 45(6), 33 38 (2002).

[147] E. V. KRISHNAMURTHY. Parallel Processing: Principles and Practice. *Addison-Wesley* (1989).

[148] V. KUMAR, A. GRAMA, A. GUPTA & G. KARYPIS. Introduction to Parallel Computing: Design and Analysis of Algorithms. *Benjamin* (1993).

[149] V. KUMAR & A. GUPTA. Analysis of scalability of parallel algorithms and architectures: a survey. in *ICS '91: Proceedings of the 5th international conference on Supercomputing*, 396–405 (ACM, New York, NY, USA, 1991).

[150] Y. LABROU, T. FININ & Y. PENG. Agent Communication Languages: The Current Landscape. *IEEE Intelligent Systems* 14(2), 45–52 (1999), DOI: http://dx.doi.org/10.1109/5254.757631.

[151] P. LACROUTE. Analysis of a parallel volume rendering system based on the shear-warp factorisation. *IEEE Trans. Visualization and Computer Graphics* 2(3), 218–231 (1996).

[152] F. LAMBERTI, C. ZUNINO, A. SANNA, A. FIUME & M. MANIEZZO. An accelerated remote graphics architecture for PDAs. *Proc. 8th Int. Conf. on 3D Web Technology* , 55–ff (2003).

[153] R. LÄMMEL. Google's MapReduce programming model — Revisited. *Sci. Comput. Program.* 68(3), 208–237 (2007), DOI: http://dx.doi.org/10.1016/j.scico.2007.07.001.

[154] L. LAMPORT. The parallel execution of DO loops. *Commun. ACM* 17(2), 83–93 (1974), DOI: http://doi.acm.org/10.1145/360827.360844.

[155] L. LAMPORT, R. SHOSTAK & M. PEASE. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4(3), 382–401 (1982), DOI: http://doi.acm.org/10.1145/357172.357176.

[156] D. LAUR & P. HANRAHAN. Hierarchical splatting: a progressive refinement algorithm for volume rendering. *ACM SIGGRAPH Computer Graphics* 25(4), 285–288 (1991).

[157] T.-Y. LEE, C. S. RAGHAVENDRA & J. B. NICHOLAS. Image composition schemes for sort-last polygon rendering on 2D mesh multicomputers. *IEEE Trans. Visualization and Graphics* 2(3), 202–217 (1996).

[158] Z. LI, H. LIU & M. PARASHAR. Enabling Autonomic Self-Managing Grid Applications. *Proceedings of SELF-START: International Workshop on Self-Properties in Complex Information Systems. Springer Verlag, Bertinoro, Italy* May-June (2004).

[159] Z. LI & M. PARASHAR. Rudder: A Rule-Based Multi-Agent Infrastructure for Supporting Autonomic Grid Applications. in *ICAC '04: Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, 278–279 (IEEE Computer Society, Washington, DC, USA, 2004).

[160] Z. LI & M. PARASHAR. Rudder: An agent-based infrastructure for autonomic composition of grid applications. *Multiagent Grid Syst.* 1(3), 183–195 (2005).

[161] LINDEN LAB. Second Life. URL: http://www.secondlife.com/, Last checked: *December 2007.*

[162] P. LINDSTROM. Out-of-core simplification of large polygonal models. *Proc. ACM SIGGRAPH* , 259–262 (2000).

[163] B. H. LISKOV & A. SNYDER. Exception Handling in CLU. *IEEE Trans. Softw. Eng.* 5(6), 546–558 (1979), DOI: http://dx.doi.org/10.1109/TSE.1979.230191.

[164] S. LOMBEYDA, L. MOLL, M. SHAND, D. BREEN & A. HEIRICH. Scalable interactive volume rendering using off-the-shelf components. in *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, 115–121 (IEEE Press, Piscataway, NJ, USA, 2001).

[165] R. F. LOPES & F. J. DA SILVA E SILVA. Fault Tolerance in a Mobile Agent Based Computational Grid. *Proc. of the sixth IEEE International Symposium on Cluster Computing and the Grid Workshops* (2006).

[166] K.-L. MA, J. S. PAINTER & M. F. KROGH. Parallel volume rendering using binary swap composition. *IEEE CG&A* 14(4), 59–67 (1994).

[167] P. MACKERRAS & B. CORRIE. Exploiting data coherence to improve parallel volume rendering. *IEEE Parallel and Distributed Technology: Systems and Applications* 2(2), 8–16 (1994).

[168] E. MANCINI, U. VILLANO, M. RAK & R. TORELLA. A Simulation-Based Framework for Autonomic Web Services. in *ICPADS '05: Proceedings of the 11th International Conference on Parallel and Distributed Systems - Workshops (ICPADS'05)*, 433–437 (IEEE Computer Society, Washington, DC, USA, 2005).

[169] D. MARQUARDT. An Algorithm for Least-Squares Estimation of Nonlinear Parameters. *SIAM Journal on Applied Mathematics* 11(431-441) (1963).

[170] R. MARSHALL, J. KEMPF, S. DYER & C. YEN. Visualization methods and simulation steering for 3D turbulence model for Lake Erie. *ACM SIGGRAPH Computer Graphics* 24(2), 89–97 (1990).

[171] N. MAZZOCCA, M. RAK & U. VILLANO. The metaPL approach to the performance analysis of distributed software systems. in *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, 142–149 (ACM, New York, NY, USA, 2002).

[172] N. MAZZOCCA, M. RAK & U. VILLANO. The Transition from a PVM Program Simulator to a Heterogeneous System Simulator: The HeSSE Project. in *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 266–273 (Springer-Verlag, London, UK, 2000).

[173] J. MCCARTHY. LISP: a programming system for symbolic manipulations. in *ACM '59: Preprints of papers presented at the 14th national meeting of the Association for Computing Machinery*, 1–4 (ACM, New York, NY, USA, 1959).

[174] M. MEISSNER, U. HOFFMANN & W. STRASSER. Enabling classification and

shading for 3D texture mapping based volume rendering using OpenGL and extensions. *Proc. IEEE Visualization* (1999).

[175] B. MELCHER & B. MITCHELL. Towards an Autonomic Framework: Self-Configuring Network Services and Developing Autonomic Applications. *Intel Technology Journal* 08(04), 279–290 November (2004).

[176] R. M. METCALFE & D. R. BOGGS. Ethernet: distributed packet switching for local computer networks. *Commun. ACM* 19(7), 395–404 (1976), DOI: http://doi.acm.org/10.1145/360248.360253.

[177] C. K. MICHAELS & M. J. BAILEY. VizWiz: A Java Applet for interactive 3D scientific visualization over the web. *Proc. IEEE Visualization* (1997).

[178] N. MINAR, M. GRAY, O. ROUP, R. KRIKORIAN & P. MAES. Hive: Distributed agents for networking things. *Proceedings of the First International Symposium on Agent Systems and Applications / Third International Symposium on Mobile Agents* , 118 (1999).

[179] P. MOCKAPETRIS. Domain Name Server RFC 1035. URL: http://www.ietf.org/rfc/rfc1035.txt, Last checked: *December 2007.*

[180] L. MOLL, M. SHAND & A. HEIRICH. Sepia: Scalable 3D Compositing Using PCI Pamette. in *FCCM '99: Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 146 (IEEE Computer Society, Washington, DC, USA, 1999).

[181] S. MOLNAR, C. M. D. ELLSWORTH & H. FUCHS. A sorting classification of parallel rendering. *IEEE CG&A* 14(4), 23–32 (1994).

[182] G. E. MOORE. Cramming more components onto integrated circuits. *Electonics* 38(8) (1965).

[183] P. J. MORAN & C. HENZE. Large field visualization with demand-driven calculation. in *VIS '99: Proceedings of the conference on Visualization '99*, 27–33 (IEEE Computer Society Press, Los Alamitos, CA, USA, 1999).

[184] K. MORELAND & D. THOMPSON. From cluster to wall with VTK. *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics* , 25–32 (2003).

[185] S. MURAKI, E. B. LUM, M. O. K.-L MA & X. LIU. A PC cluster system for simultaneous interactive volumetric modeling and visualization. *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics* , 95–102 (2003).

[186] M. NAEF, E. LAMBORAY, O. STAADT & M. GROSS. The blue-c distributed scene graph. *Proc. IEEE Virtual Reality* , 275–276 (2003).

[187] J. NEIDER & T. DAVIS. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, (1993).

[188] NEXT & SUN MICROSYSTEMS. OpenStep API - property lists. URL: http://developer.apple.com/documentation/Cocoa/Conceptual/PropertyLists/PropertyLists.html, (1994).

[189] NEXT & SUN MICROSYSTEMS. OpenStep Specification. URL: http://www.gnustep.org/resources/OpenStepSpec/OpenStepSpec.html, (1994).

[190] A. NORTON & A. ROCKWOOD. Enabling view-dependant progressive volume visualization on the grid. *IEEE CG&A* 23(2), 22–31 (2003).

[191] M. OGATA, S. MURAKI, X. LIU & K.-L. MA. The design and evaluation of a pipelined image compositing device for massively parallel volume rendering. in *VG '03: Proceedings of the 2003 Eurographics/IEEE TVCG Workshop on Volume graphics*, 61–68 (ACM, New York, NY, USA, 2003).

[192] M. PARASHAR *et al.* AutoMate: Enabling Autonomic Applications on the Grid. *Cluster Computing* 9(2), 161–174 (2006), DOI: http://dx.doi.org/10.1007/s10586-006-7561-5.

[193] M. S. G. PARKER, C. D. H. MILLER & C. R. JOHNSON. An integrated problem solving environment: the SCIRun computational steering system. *Proc. 31st Hawaii Int. Conf. on System Sciences* (1998).

[194] S. G. PARKER & C. R. JOHNSON. SCIRun: a scientific programming environment for computational steering. in *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, 52 (ACM, New York, NY, USA, 1995).

[195] J. PASCOE, N. RYAN & D. MORSE. Using while moving: HCI issues in fieldwork environments. *ACM Trans. Computer-Human Interaction* 7(3), 417–437 (2000).

[196] H. PENG, H. XIONG & J. SHI. Parallel-SG: research of parallel graphics rendering system on PC-Cluster. in *VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, 27–33 (ACM, New York, NY, USA, 2006).

[197] W. P. PETERSEN. Vector Fortran for numerical problems on CRAY-1. *Commun. ACM* 26(11), 1008–1021 (1983), DOI: http://doi.acm.org/10.1145/182.358469.

[198] H. PFISTER, J. HARDENBERGH, J. KNITTEL, H. LAUER & L. SEILER. The VolumePro real-time ray-casting system. *Proc. ACM SIGGRAPH* , 251–260 (1999).

[199] H. PFISTER & A. KAUFMAN. Cube-4 – a scalable architecture for real-time volume rendering. *Proc. ACM/IEEE Symp. Volume Rendering* , 47–54 (1996).

[200] H. PFISTER, M. ZWICKER, J. V. BAAR & M. GROSS. Surfels: surface elements as rendering primitives. *Proc. ACM SIGGRAPH*, 335–342 (2000).

[201] W. PIERCE. Failure-Tolerant Computer Design. *Academic Press, New York* (1965).

[202] R. PINKHAM, M. NOVAK & K. GUTTAG. Video RAM excels at fast graphics. *Electronic Design* 31(17), 161–182 (1983).

[203] J. POSTEL. Internet Control Message Procotol (ICMP). URL: `http://tools.ietf.org/html/rfc792`, (1981), Last checked: *December 2007.*

[204] QWAQ. Qwaq Forum. URL: `http://www.qwaq.com/qwaq_forums.html`, (2006), Last checked: *December 2007.*

[205] B. RANDELL. System structure for software fault tolerance. in *Proceedings of the international conference on Reliable software*, 437–449 (ACM, New York, NY, USA, 1975).

[206] B. RANDELL, P. LEE & P. C. TRELEAVEN. Reliability Issues in Computing System Design. *ACM Comput. Surv.* 10(2), 123–165 (1978), DOI: `http://doi.acm.org/10.1145/356725.356729`.

[207] R. V. RENESSE, K. P. BIRMAN & W. VOGELS. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.* 21(2), 164–206 (2003), DOI: `http://doi.acm.org/10.1145/762483.762485`.

[208] R. V. RENESSE, Y. MINSKY & M. HAYDEN. A Gossip-Style Failure Detection Service. Technical report, Ithaca, NY, USA, (1998).

[209] C. REZK-SALAMA, K. ENGEL, M. BAUER, G. GREINER & T. ERTL. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage-rasterization. *Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 109–118 (2000).

[210] C. REZK-SALAMA, P. HASTREITER, C. TEITZEL & T. ERTL. Interactive exploration of volume line integral convolution based on 3D-texture mapping. *Proc. Visualization*, 233–240 (1999).

[211] M. RIDING *et al.* e-viz: Towards an integrated framework for high performance visualization. *UK e-Science All Hands Meeting 2005*, 1026–1032 (2005).

[212] N. ROARD & M. W. JONES. Agents based visualization and strategies. *in Proceedings of WSCG 2006*, 63–70 (2006).

[213] S. ROETTGER, S. GUTHE, D. WEISKOPF, T. ERTL & W. STRASSER. Smart hardware accelerated volume rendering. *Proc. Eurographics/IEEE-TCVG Symp. Visualization* (2003).

[214] A. ROTEM-GAL-OZ. Fallacies of Distributed Computing Explained. URL: http://www.rgoarchitects.com/Files/fallacies.pdf, (2006), Last checked: *December 2007.*

[215] D. L. RUSSELL. State Restoration in Systems of Communicating Processes. *IEEE Trans. Softw. Eng.* 6(2), 183–194 (1980), DOI: http://dx.doi.org/10.1109/TSE.1980.230469.

[216] S. J. RUSSELL & P. NORVIG. *Artificial Intelligence: A Modern Approach.* Pearson Education, (2003).

[217] H. SAMET. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.* 16(2), 187–260 (1984), DOI: http://doi.acm.org/10.1145/356924.356930.

[218] P. B. SCHNECK. Automatic recognition of vector and parallel operations in a higher level language. in *ACM '72: Proceedings of the ACM annual conference,* 772–779 (ACM, New York, NY, USA, 1972).

[219] W. SCHROEDER, K. MARTIN & B. LORENSEN. The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics. *Prentice Hall, second edition* (1998).

[220] J. SHALF & E. W. BETHEL. The Grid and future visualization systems architectures. *IEEE CG&A* 23(2), 6–9 (2003).

[221] J. A. SHARP (ED.). Data Flow Computing: Theory and Practice. *Ablex Publishing* (1992).

[222] Y. SHI. Reevaluating Amdahl's Law and Gustafson's Law. URL: http://www.cis.temple.edu/~shi/docs/amdahl/amdahl.html, (1996).

[223] K. SISTLA, A. D. GEORGE & R. W. TODD. Experimental Analysis of a Gossip-Based Service for Scalable, Distributed Failure Detection and Consensus. *Cluster Computing* 6(3), 237–251 (2003), DOI: http://dx.doi.org/10.1023/A:1023592621046.

[224] D. B. SKILLICORN. A Taxonomy for Computer Architectures. *IEEE Computer* 21(11), 46–57 (1988).

[225] D. B. SKILLICORN & D. TALIA. Models and languages for parallel computation. *ACM Computing Surveys* 30, 123–169 (1998).

[226] B. C. SMITH. Reflection and semantics in LISP. in *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages,* 23–35 (ACM, New York, NY, USA, 1984).

[227] D. SONG & E. GOLIN. Fine-grain visualization algorithms in dataflow environments. *Proc. IEEE Visualization* , 126–133 (1993).

[228] G. L. STEELE. Common Lisp: the language. *Digital Press, 2nd edition* (1990).

[243] T. TERASAWA, S. OGURA, K. INOUE & H. AMANO. A cache coherency protocol for multiprocessor chip. *Proc. 7th IEEE Int. Conf. on Wafer Scale Integration*, 238–247 (1995).

[244] G. TESAURO *et al.* A Multi-Agent Systems Approach to Autonomic Computing. in *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, 464–471 (IEEE Computer Society, Washington, DC, USA, 2004).

[245] THE FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS. FIPA97 Specifications. URL: `http://www.fipa.org/`, (1997), Last checked: *December 2007.*

[246] TROLLTECH. Qt Concurrent. URL: `http://labs.trolltech.com/blogs/category/labs/threads/qt-concurrent/`, (2007).

[247] S. TUECKE *et al.* Open Grid Services Infrastructure (OGSI) Version 1.0. *Global Grid Forum Draft Recommendation* (2003).

[248] C. UPSON *et al.* The application visualization system: A computational environment for scientific visualization. *IEEE CG&A* 9(4), 30–42 (1989).

[249] P. URBAN, X. DEFAGO & A. SCHIPER. Chasing the FLP impossibility result in a LAN, or how robust can a fault tolerant server be. *IEEE Symposium on Reliable Distributed Systems*, 190–193 (2001), URL: `citeseer.ist.psu.edu/urban01chasing.html`.

[250] S. URBANEK. Steptalk. URL: `http://www.gnustep.org/experience/StepTalk.html`, (2003).

[251] L. G. VALIANT. A bridging model for parallel computation. *Commun. ACM* 33(8), 103–111 (1990), DOI: `http://doi.acm.org/10.1145/79173.79181`.

[252] D. C. VANDERSTER, N. J. DIMOPOULOS & R. J. SOBIE. Intelligent Selection of Fault Tolerance Techniques on the Grid. *Third IEEE International Conference on e-Science and Grid Computing* (2007).

[253] J. VEENSTRA & N. AHUJA. Line drawings of octree-represented objects. *ACM Trans. Graphics* 7(1), 61–75 (1988).

[254] W. E. WALSH, G. TESAURO, J. O. KEPHART & R. DAS. Utility functions in autonomic systems. *Proc. 1st Int. Conf. on Autonomic Computing* (2004).

[255] T. WATANABE & A. YONEZAWA. Reflection in an object-oriented concurrent language. in *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications*, 306–315 (ACM, New York, NY, USA, 1988).

[256] M. WEISER. Some computer science issues in ubiquitous computing. *Communications of the ACM* 36(7), 75–84 (1993).

[257] D. WERTHIMER, J. COBB, M. LEBOFSKY, D. ANDERSON & E. KORPELA. SETI@HOME—massively distributed computing for SETI. *Comput. Sci. Eng.* 3(1), 78–83 (2001), DOI: http://dx.doi.org/10.1109/5992.895191.

[258] R. WESTERMANN & T. ERTL. Efficiently using graphics hardware in volume rendering applications. *Proc. ACM SIGGRAPH* , 169–177 (1998).

[259] D. WINER. XML-RPC specification. URL: http://www.xmlrpc.com/spec, June (1999).

[260] C. M. WITTENBRINK & M. HARRINGTON. A scalable MIMD volume rendering algorithm. *Proc. 8th Int. Parallel Processing Symp.* , 916–922 (1994).

[261] M. WOLF, Z. CAI, W. HUANG & K. SCHWAN. Smartpointers: personalized scientific data portals in your hand. *In Proc. ACM/IEEE Conf. on Supercomputing* , 1–16 (2002).

[262] J. WOOD, K. BRODLIE & J. WALTON. gViz – visualization and steering for the Grid. *Proc. e-Science All Hands Meeting* (2003).

[263] J. WOOD, H. WRIGHT & K. BRODLIE. Collaborative Visualization. *Proc. IEEE Visualization* , 253–259 (1997).

[264] J. D. WOOD, K. W. BRODLIE & H. WRIGHT. Visualization over the world wide web and its application to environmental data. *Proc. IEEE Visualization* , 81–86 (1996).

[265] J. D. WOOD & H. WRIGHT. Steering via the image in local, distributed and collaborative Settings. *in Proceedings of the UK e-Science All Hands Meeting 2005* , 1026–1032 (2005).

[266] B. WOODCOCK & B. MANNING. Multicast Domain Name Service. *Internet Engineering Task Force* (2000).

[267] M. WOOLDRIDGE. Agent-based software engineering. *IEE Proc. Software Engineering* 144(1), 26–37 (1997).

[268] M. WOOLDRIDGE & N. R. JENNINGS. Intelligent agents: Theory and practice. *Knowledge Engineering Review* 10(2), 115–152 (1995).

[269] H. WRIGHT, K. BRODLIE & T. DAVID. Navigating High-dimensional Spaces to Support Design Steering. *Proceedings of IEEE Visualization 2000* , 291–296 (2000).

[270] B. WYLIE, C. PAVLAKOS, V. LEWIS & K. MORELAND. Scalable rendering on PC clusters. *IEEE CG&A* 21(4), 62–69 (2001).

[271] G. YOO, J. PARK & E. LEE. Hybrid Prediction Model for improving Reliability in Self-Healing System. in *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, 108–116 (IEEE Computer Society, Washington, DC, USA, 2006).

[272] X. ZHANG & X. QIN. Performance prediction and evaluation of parallel processing on a NUMA multiprocessor. *IEEE Trans. Software Engineering* 17(10), 1059 (1991).

[273] A. F. ZORZO & F. R. MENEGUZZI. An agent model for fault-tolerant systems. *Proceedings of the 2005 ACM Symposium on Applied Computing* , 60–65 (2005).